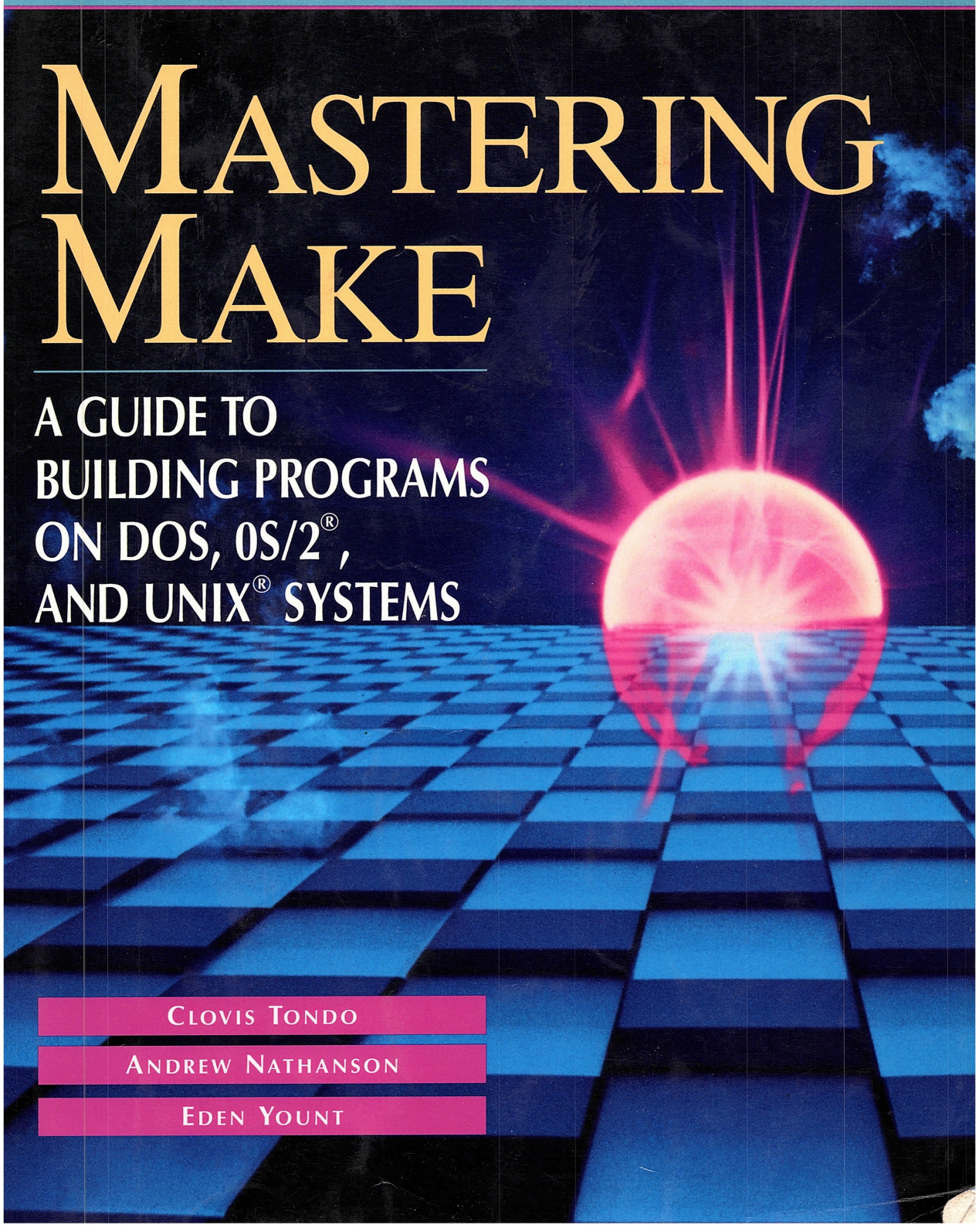
SECOND EDITION

MASTERING MAKE

A GUIDE TO BUILDING PROGRAMS ON DOS, OS/2®, AND UNIX® SYSTEMS



CLOVIS TONDO

ANDREW NATHANSON

EDEN YOUNT

Mastering MAKE

A Guide to Building Programs
on DOS, OS/2, and UNIX Systems

Second Edition

Clovis L. Tondo
T&T TechWorks, Inc.
Coral Springs, Florida

Andrew Nathanson
Independent Software Engineer
Parkland, Florida

Eden Yount
Consultant in Information Engineering
Merritt Island, Florida

P T R Prentice Hall
Englewood Cliffs, New Jersey 07632

Tondo, Clovis L.
Mastering MAKE, a guide to building programs on DOS, OS/2, and UNIX systems / Clovis L. Tondo, Andrew Nathanson, Eden Yount.
p. cm.
Includes index.
ISBN 0-13-121906-5
1. Utilities (Computer programs) 2. Make (Computer file) 3. Computer software–Development–Management I. Nathanson, Andrew. II. Yount, Eden. III. Title.
QA76.76.U84T65 1994 94-4529
005.13'3–dc20 CIP

Editorial/production supervision: *Ann Sullivan*
Cover designer: *Design Solutions*
Cover photo credit:*The Image Bank©Pete Turner*
Manufacturing manager: *Alexis R. Heydt*
Composition: *T&T TechWorks, Inc.*
Acquisitions editor: *Gregory G. Doench*
Editorial assistant: *Marcy Levine*

©1994 by P T R Prentice Hall
Prentice-Hall, Inc.
A Paramount Communications Company
Englewood Cliffs, New Jersey 07632

The publisher offers discounts on this book when ordered in bulk quantities. For more information, contact:
Corporate Sales Department; P T R Prentice Hall; 113 Sylvan Avenue;
Englewood Cliffs, New Jersey 07632 Phone: (201) 592-2863 FAX: (201) 592-2249

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

MS-DOS, Microsoft C, and Windows are trademarks of Microsoft Corporation. IBM, OS/2, Presentation Manager, and C Set/2++ are trademarks of International Business Machines Corporation. Borland C++ is a trademark of Borland International. UNIX is a trademark of UNIX System Laboratories, Inc. X Windows System is a trademark of The Massachusetts Institute of Technology. Motif is a trademark of the Open Software Foundation. LaTeX is a trademark of Addison-Wesley. TeX is a trademark of the American Mathematical Society.

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-121906-5

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Contents

Preface ix

Preface to the First Edition xiii

1 Fundamentals of MAKE 1
- The Theory Behind MAKE . . . 2
- The Makefile . . . 2
  - Syntax Rules . . . 2
  - Targets . . . 4
  - Dependencies . . . 5
- MAKE Operations . . . 6
- Summary . . . 8

2 Invoking MAKE 11
- Common MAKE Options . . . 11
- Examples . . . 12
- Using Hard-Coded Commands . . . 13
- Using Commands with Macros . . . 14
- More About Dependencies . . . 17
- Summary . . . 18

3 More About MAKE 21
- More About Macros . . . 21
  - User-Defined Macros . . . 22
  - Predefined Macros . . . 23
  - Macros and the p Option . . . 26
- Pseudotargets . . . 26
  - User-Defined Pseudotargets . . . 27
  - Predefined Pseudotargets . . . 27

Inference Rules and Implicit Commands 29
User-Defined Inference Rules 31
Predefined Inference Rules 34
How Are We Doing? 36
Application: Makefile for Text Formatting 38
Summary 40

4 Advanced Topics 45
Using MAKE Recursively 45
Using Environment Variables in Makefiles 52
Macro Information Substitution 58
Beware the Macro Happy Makefile 59
Common Errors 61
Summary 64

5 The MAKE Utility on DOS Systems 67
NMAKE on DOS 67
Invoking NMAKE 68
Command Line Options 68
Preprocessing Directives 69
Expressions in Preprocessing 75
Special Cases 78
Creating Dependencies Dynamically 80
Input Files and In-Line Response Files 84
Using Command Level Processing in Makefiles 87
NMAKE and the tools.ini File 89
Macro Precedence 90
Using NMAKE to Develop GUI Applications 91
Borland's MAKE on DOS 97
Invoking MAKE 97
Command Line Options 98
Preprocessing Directives 99
Special Cases 107
Creating Dependencies Dynamically 107
Input Files and In-Line Response Files 110
Using Command Level Processing in Makefiles 114
MAKE and the builtins.mak File 116
Macro Precedence 118

Using MAKE to Develop GUI Applications 118
Summary 125

6 The MAKE Utility on OS/2 Systems 131
NMAKE on OS/2 131
Invoking NMAKE 132
Command Line Options 132
Preprocessing Directives 133
Expressions in Preprocessing 138
Special Cases 141
Creating Dependencies Dynamically 143
Input Files and In-Line Response Files 147
Using Command Level Processing in Makefiles 150
NMAKE and the tools.ini File 151
Macro Precedence 152
Using NMAKE to Develop GUI Applications 153
Borland's MAKE on OS/2 159
Invoking MAKE 159
Command Line Options 160
Preprocessing Directives 161
Special Cases 169
Creating Dependencies Dynamically 170
Input Files and In-Line Response Files 172
Using Command Level Processing in Makefiles 176
MAKE and the builtins.mak File 178
Macro Precedence 179
Using MAKE to Develop GUI Applications 180
Summary 186

7 UNIX System's MAKE 193
Invoking MAKE 193
Command Line Options 194
Makefile Command Lines 194
Environment Variables 195
Macros and Macro Precedence 197
Suffixes and Inference Rules 200
About SCCS 201
Using MAKE to Develop GUI Applications 206

Summary 207

A Comparing MAKE 209

B Microsoft's NMAKE 211

C IBM's NMAKE 215

D Borland's MAKE 219

E UNIX System's MAKE 223

F Windows Sample 227
The Program 227
The .rc File 234
The .dlg File 235
Makefile for Microsoft's NMAKE 236
Makefile for Borland's MAKE 239

G Presentation Manager Sample 243
The Program 243
The .rc File 251
The .dlg File 252
Makefile for IBM's NMAKE 253
Makefile for Borland's MAKE 256

H X Windows Sample 259
The Program 259
The Makefile for UNIX's MAKE 268

I Depends Utility 269
Purpose 269
Syntax 269
Description 269
Examples 270
Observations 271
Outline for the depends Utility 272
The Program 273
Makefile for depends Using Microsoft's NMAKE 291
Makefile for depends Using Borland's MAKE 295

Index 301

Preface

The first edition of *Mastering MAKE* was targeted toward what we believed was a wide audience. Our readers have since let us know that the potential audience is even broader than we initially thought. And, of course, the computer industry has not stood still for the past two years. So we returned to *Mastering MAKE* with the goals of expanding and updating the content and improving the presentation.

In revising the contents for this second edition of *Mastering MAKE*, we have maintained our original purpose: to provide a basic, yet comprehensive explanation of the usefulness and subtleties of the *make* utility in general and to demonstrate the concepts in action with numerous examples for specific versions of *make* running under several popular operating systems.

We have expanded the scope of information in several ways. First, this edition covers five versions of *make* on three operating systems (compared to three versions on two operating systems in the first edition). Second, acknowledging the growing number of software developers and compiler manufacturers who are moving to a graphical user interface (GUI), we explain how to create GUI applications using *make*. Thus, in this second edition of *Mastering MAKE* you will find discussions of:

- NMAKE from Microsoft and MAKE from Borland on DOS systems
- NMAKE from IBM and MAKE from Borland on OS/2 systems
- MAKE on UNIX systems

You will also find explanations of how GUI programs are created for:

- Windows on DOS systems
- Presentation Manager on OS/2 systems
- X Windows on UNIX systems

Numerous, tested examples support all these topics and help to speed your progress toward effectively applying *make* to your own software development projects. We have, of course, used the most current compilers (at the time of writing) to develop and test our examples.

As in the first edition, we use NMAKE from Microsoft as our baseline version. The `nmake` command that is featured in many of our examples is part of the C/C++ Compiler, Version 7.0 package from Microsoft. This utility has been a part of the Microsoft family of products for several years.

In the OS/2 environment, the `nmake` command is part of the OS/2 Toolkit Version 2.1 package from IBM. This product is licensed from Microsoft. It works in conjunction with the Workframe/2 and IBM C Set/2++ compiler and supports 32-bit processing.

Borland's product MAKE is available for both the DOS and OS/2 environments. The `make` command is part of the Borland C++ Compiler, Version 3.1 for DOS and Version 1.0 for OS/2. The OS/2 version is a 32-bit utility.

On UNIX systems, the `make` command is part of the system software package. Examples in this book identified for use on UNIX systems have been tested on a system based on UNIX System V.3.

As you would expect, differences exist among the various versions of *make*. We define the variations wherever the IBM, Borland, or UNIX version differs from the Microsoft version. The example makefiles in the book are written for use with Microsoft's `nmake` command unless specifically stated otherwise.

This is a good place to clear up some possible confusion or misunderstanding. The order in which the material is presented is purely arbitrary. It should not be construed as a reflection of either our personal preferences or our opinions as to the relative merit of these or any other products.

To accommodate the expanded scope of the book, we have made some changes to its organization. As in the first edition, Chapter 1 begins with the principal concepts of *make* and introduces the basic elements of a makefile. Chapters 2 and 3 build on this foundation in logical, easy-to-follow steps. Chapter 4 moves on to advanced topics that will lead you to write more sophisticated makefiles.

The detailed uses of each of the five versions of *make* are addressed in Chapters 5, 6, and 7. It is here that the structure of this edition diverges somewhat from the first. Chapter 5 presents the use of *make* on DOS systems, focusing on NMAKE from Microsoft, MAKE from Borland, and Microsoft's GUI, Windows. In Chapter 6 we turn to OS/2 systems, covering

NMAKE from IBM, MAKE from Borland, and IBM's GUI, Presentation Manager. Finally, Chapter 7 details the use of MAKE on UNIX systems, enhanced with information about X Windows.

Finally, we revisited the design of the book. We decided to keep some elements as they were in the first edition and to change others, with an eye toward improving the overall appearance and readability of this new edition. In particular, the typographic conventions are the same as before:

- *make* represents the generic name of the utility, regardless of the product name or operating system
- `nmake` and `make` represent what you enter on the command line
- NMAKE and MAKE are the names of the individual products

The source code and makefiles for Windows, Presentation Manager, X Windows with Motif, and the `depends` program are available via anonymous ftp from Prentice Hall at `ftp.prenhall.com`; the code appears in the directory `pub/software/MasteringMAKE`, and it contains the source code for Appendix F (Windows), Appendix G (Presentation Manager), Appendix H (X Windows with Motif), and Appendix I (the `depends` program).

With its expanded scope and improved design, this second edition of *Mastering MAKE* – like its predecessor – allows us to share with you what we've learned about this valuable programming tool. We are confident that by the time you have progressed through the combined tutorial and reference material in these pages, your mastery of *make* will help you to more effectively manage your software development projects.

Acknowledgments

To the uninitiated, producing the second edition of an existing work might seem a relatively simple task. Would that it were so! But the publication of any book, regardless of the edition, requires the effort of a great many people besides the authors.

Jointly, we appreciate the constructive criticism, technical support, and motivation generously provided by so many — at Borland International, Karen Giles; at IBM, Edson Ferretti, David Hoffman, and Red Viscuso; at Keane Inc., Paul van der Meulen; at Pencom Systems, Tom Bagli; at Y&Y Inc., Louis Vosloo; at Nova Southeastern University, Ted Flynn, George

Gabb, Daryl Hulce, Jennifer Isaacs, Ginny McLain, and Raul Salazar; and at Florida Atlantic University, Neal Coulter, George Edmonds, Tom Horton, Sam Hsu, and Maria Petri.

The talented and enthusiastic team at Prentice Hall have once again brought all the pieces together to make this second edition of *Mastering MAKE* a reality. In particular, we acknowledge the invaluable contributions of Greg Doench, Marcy Levine, Sophie Papanikolaou, and Ann Sullivan.

Individually, we are indebted to several others who have contributed in personal and special ways to this effort.

Clovis thanks Caren Tondo for her patience and sense of humor and is grateful for the support of Luiz and Elizabete Biavatti, Harriet Blatman, Bill Dyer, Ed and Sandy Gorley, Lauren Marshall, Nelson Felippe da Silva, Claire and João Vendruscolo, Denise and Bob Webster, and Joyce and Bob Webster.

Andrew thanks Lorraine Patrone, Elaine Nathanson, and the Guillets (Abbe, Dominique, Philip and Stephanie) and rededicates his efforts to the loving memory of his father, Joseph Nathanson.

Eden is grateful for the support and encouragement of Wayne and Margaret Yount; mentors and colleagues Dan Grostick, Tom Cianflone, Debbie Richards, Paula Devereaux, and Marilyn Johnson; and above all, John Gallipoli who makes us laugh and teaches us to believe in ourselves.

CLT
AN
EY

Preface to the First Edition

The purpose of this book is to help you, as a programmer, understand and use the *make* utility. If you spend much of your time, as we do, in a software development environment, you probably are well aware of the need for a tool such as *make*. If you are just entering this challenging arena — or are drawn by simple curiosity — you soon will appreciate the value of this utility.

It has been our experience that a good many programmers have heard of, and even worked with, the *make* utility without ever understanding exactly what it does or why they are (or should be) using it. And while many programmers write makefiles that work, they do not exploit the full power of the tool. Others (including even veteran programmers) struggle along in the too common, mostly unproductive trial-and-error process of trying to learn how to use *make* effectively. And they enjoy little success for their efforts.

Other than the usual manuals and online help files that come with a DOS compiler or a UNIX system, there is a paucity of information available about this invaluable tool. What is available seems directed to a narrow audience and intended primarily as reference material, without offering much in the way of guidance to the uninitiated.

Mastering MAKE is our effort to fill this void:

- to explain how *make* works
- to suggest reasons for using it
- to demonstrate its versatility and flexibility in a variety of real-life applications
- to provide a smooth, logical progression for learning the utility from its basic concepts to its most sophisticated implementations

In short, to share with you what we've learned.

We approached this project with the idea of writing a book that would benefit a wide audience. We wanted to address the needs of less experienced programmers while augmenting the knowledge already acquired by more experienced software developers.

At the same time, we did not want to limit our discussion to a single version of *make*. Long established on UNIX systems, the *make* utility is available — though perhaps less widely known or used — on other operating systems as well. We wanted to cover not only universal features of *make*, but also those that apply specifically to particular operating systems: namely, DOS and UNIX systems.

It was a challenge that we set for ourselves, but one that we believe we have met.

We have selected three versions of *make* around which to develop both general principles and specific applications. These versions are:

- NMAKE from Microsoft

  The `nmake` command is part of the C Compiler, Version 6.0A package from Microsoft. It is a relatively new command and is not part of any previous C language product from Microsoft.

- MAKE from Borland

  The `make` command is part of the Turbo C++ Compiler, Version 2.0, which (at the time of writing) is the latest of the Turbo C++ packages from Borland.

- MAKE for UNIX systems

  The `make` command is part of UNIX systems. Examples in this book identified for use on UNIX systems have been tested on a system based on UNIX System V.3.

We have used NMAKE from Microsoft (with C version 6.0A) as our baseline version of *make*. Where differences exist for MAKE from Borland or for MAKE on UNIX systems, the variations are identified. Example makefiles are written for use with the `nmake` command, unless specifically stated otherwise.

Of course, we must have some common ground from which to start. In the present case, we assume you are comfortably familiar with your operating system environment, whether you are working on a DOS or a UNIX system.

(However, you are not required to have achieved the status of guru!) We assume also that you have a comfortable working relationship with your compiler. With this foundation, you should be able to progress painlessly through these pages, regardless of your level of experience with software development projects or the *make* utility.

You'll find that *Mastering MAKE* combines elements of both a tutorial and a reference book. With the introduction of each new concept, you will find examples of tested makefiles. You can use these to gain confidence and to test your understanding. At the end of each chapter we review salient concepts. These summaries are useful as reference material, particularly for the more experienced programmers in our audience. We have designed the appendices to serve as convenient quick reference guides; you'll be able to identify effortlessly the features that apply specifically to each version of *make*.

We begin in Chapter 1 with the principal concepts of *make* and introduce the basic elements of a makefile, then build on these in Chapters 2 and 3 in logical, easy-to-follow steps. By the time you reach Chapter 4, you will be ready to take full advantage of the power of *make* as you develop sophisticated makefiles to support practical, real-life applications. At every step, we emphasize good programming practices and the value of developing makefiles that are flexible and easy to maintain.

Beginning with Chapter 5, we look at the specifics of each version of *make*. Chapter 5 is devoted to Microsoft's NMAKE, Chapter 6 to Borland's MAKE, and Chapter 7 to MAKE on UNIX systems.

Inevitably, some of the information in *Mastering MAKE* duplicates information provided in the manuals and online help files that came with your DOS compiler or your UNIX system. *Mastering MAKE* is not meant to supplant that documentation, but to supplement it with tactics and strategies acquired from our combined experiences with the utility. Even those among you who are well-versed in the intricacies of *make* will undoubtedly pick up some subtleties and hints that may have escaped you in the past.

Regardless of your individual starting point, we are confident you will find, at the end, that you have indeed mastered *make*.

Acknowledgments

The writing and publishing of any book is an undertaking that requires the efforts of many people, not just the authors. We have a number of people

to whom we owe our appreciation. Jointly, we thank the reviewers, who offered insightful suggestions for improving the book: Richard Fencel, Mary Forlenza, George Gabb, Tom Harvey, Bruce Leung, Carlos Tondo, and an anonymous reviewer. We thank also Jeff Codella, Dave Egan, David Frith, and David Thomas for their help with hardware.

Separately, we owe thanks to individuals who provided support and encouragement to each of us.

Clovis thanks Caren Webster, Klaus Luedtke, Irma Miller, Carl and Sandy Schuster, and Bob Webster for their support.

Andrew thanks Lorraine Patrone, David Hoffman, the Guillets, and Elaine Nathanson and dedicates his efforts to the memory of Joseph Nathanson.

Eden thanks Paula Devereaux and, especially, John Gallipoli, whose patience and sense of humor make all things possible.

Finally, we are grateful to the people at Prentice Hall whose various talents and skills helped guide and shape this project from concept through completion. In particular, we thank Greg Doench, Joan Magrabi, and Mary Rottino without whom *Mastering MAKE* would never have become a reality.

CLT
AN
EY

Chapter 1

Fundamentals of MAKE

The *make* utility was created to help programmers manage software development projects and files associated with those projects. The original version of *make* (created by Stuart I. Feldman) runs on UNIX systems. A few versions of *make* exist for DOS and OS/2.

In this book, we address two versions of *make* on DOS and two versions on OS/2, as well as MAKE for UNIX systems. The DOS versions we discuss are NMAKE by Microsoft and MAKE by Borland; the OS/2 versions are NMAKE by IBM and MAKE by Borland. To simplify the discussion, we will refer to the DOS and OS/2 versions as the PC versions of *make*.

To understand how *make* operates, let's first discuss the need for such a tool. Here is a typical scenario:

> You are working on a number of programs that will be compiled and linked together to form a single executable program. Every time you change one of the files, you have to recompile that file, then relink it with the existing object files to create a new executable program.

This scenario presents a significant problem: you, the programmer, have to remember which files have been modified and, consequently, which files have to be recompiled. Because you need to remember many details, keeping track of modified files and recompiling them soon become cumbersome, difficult tasks.

The *make* utility handles all the details of compiling the modified files for you. To accomplish the task, *make* follows a set of rules that you provide in a special file called a makefile.

The Theory Behind MAKE

The theory behind *make* is straightforward: *make* compares the date and time of a source file with the date and time of the associated object file. (Together, the date and time are called the time stamp.) If the comparison shows that the source file is newer than the object file — or if the object file does not exist — *make* performs the tasks you have listed in the makefile to convert the source file into an object file. On the other hand, if the object file is newer than the source file, then *make* recognizes that it does not have to recompile the source file.

The Makefile

We mentioned earlier that *make* follows a set of rules that you provide in a special file called a makefile. A makefile consists of several elements. Two elements of a makefile are targets and dependencies; we meet them in this chapter. Other elements, such as commands, macro definitions, and inference rules, are covered in later chapters.

Syntax Rules

Here is our first makefile (the example uses Microsoft's C compiler):

```
prog.exe:  prog1.obj  prog2.obj
        cl  -o  prog.exe  prog1.obj  prog2.obj

prog1.obj:  prog1.c
        cl  -c  -W2  -O  prog1.c

prog2.obj:  prog2.c
        cl  -c  -W2  -O  prog2.c
```

We must follow a few rules when we write a makefile. The example we just looked at illustrates the first two rules. Our makefile consists of three similar pairs of lines. In each set, the first line is a target/dependency line, and the second is a command line. Here are the two rules for these types of lines:

1. A target/dependency line must begin in the first column of the line.

2. A command line must be indented.

The requirements for indenting a command line vary depending on the version of *make*. With the PC versions, we can use blank spaces or tabs; one blank space or one tab is the required minimum. However, MAKE on UNIX systems requires a tab — and not spaces — at the beginning of the command line.

Our first makefile is very simple. We'll see later, in more complex examples, instances in which a list of dependencies or a command is so long that it does not fit on one line but must be carried over to a second line, or more.

This situation gives us our third syntax rule:

3. We must use a backslash (\) at the end of a line to tell *make* that the next line is a continuation of the current line.

We can also use continuation lines to enhance the readability of the makefile. We will see this technique in practice in later examples.

Good coding practice applies as much to the makefiles we write as to any other type of program we develop. This means that, except for very simple cases (such as our first example), we should include comments to explain the purpose of individual lines or groups of lines in a makefile. Our fourth syntax rule applies to comments in a makefile:

4. We must use a pound sign (#) to tell *make* that what follows is a comment and not a makefile statement to be acted on.

We can put a comment on a line by itself, or we can put it on a line that *make* executes. When we put a comment on a line by itself, the pound sign can be anywhere on the line, although common practice is to put it in the first column. When we put a comment on the same line as an executable makefile statement, the comment goes at the end of the line; that is, the pound sign follows the makefile statements and everything that follows the pound sign is considered to be a comment, which *make* ignores.

Comment lines cannot be continued with the backslash character. However, we can write multiple comment lines anywhere in the makefile as long as we start each comment with a pound sign.

Targets

A target is a name followed by a colon (:). The name appears at the beginning of a line. (The other names on the line are called dependencies. We will tell you about dependencies a little later.)

In our makefile, the first line

```
prog.exe:  prog1.obj  prog2.obj
```

begins with the target `prog.exe`. Our makefile has two more targets. The third line

```
prog1.obj:  prog1.c
```

begins with the target `prog1.obj`. And on the fifth line

```
prog2.obj:  prog2.c
```

the target is `prog2.obj`.

The default is for *make* to build the first target in the makefile. When we enter

```
nmake
```

make understands that we want to build the target `prog.exe`. (Use `make`, instead of `nmake`, if you are using Borland's MAKE or MAKE on a UNIX system.)

We can tell *make* explicitly which target we want to build by issuing, for example:

```
nmake  prog.exe
```

This command is equivalent to the plain `nmake` command we just looked at because `prog.exe` is the first target in the makefile. Unless we tell it otherwise, *make* builds the first target.

Here is another example:

```
nmake  prog2.obj
```

This command tells *make* to build only the target `prog2.obj`. In this case, *make* concerns itself only with these lines of our makefile:

```
prog2.obj:  prog2.c
        cl  -c  -W2  -O  prog2.c
```

The target `prog2.obj` depends on `prog2.c`. If `prog2.c` is newer than `prog2.obj`, or if `prog2.obj` does not exist, *make* issues the command

```
cl  -c  -W2  -O  prog2.c
```

which builds the target `prog2.obj`. Then *make* stops because no other dependencies are associated with the target `prog2.obj`.

Each target can have its own set of one or more explicit commands; that is, commands that appear explicitly with the target to which they apply. (Later we will introduce you to implicit commands.)

Dependencies

Two types of dependencies occur in a makefile: direct dependencies and indirect dependencies.

A direct dependency appears with the target. This means the target depends on the file or files listed after the colon.

In our makefile, the target `prog.exe` has two direct dependencies, `prog1.obj` and `prog2.obj`:

```
prog.exe:  prog1.obj  prog2.obj
```

The direct dependency `prog1.obj`, in turn, depends on another file:

```
prog1.obj:  prog1.c
```

`prog1.c` is a direct dependency for the target `prog1.obj`. The file `prog1.obj` is a direct dependency for `prog.exe`. Consequently, `prog1.c` is an indirect dependency for `prog.exe`.

An indirect dependency is indirectly related to the first target.

Let's look again at our makefile example for a clearer understanding of direct and indirect dependencies. If we draw out the order of precedence, we end up with a tree that shows the dependencies for the target `prog.exe`. See Figure 1.1 to observe how *make* views the dependencies.

The first target `prog.exe` depends directly on the files `prog1.obj` and `prog2.obj`. `prog1.obj` depends directly on `prog1.c`, and `prog2.obj` depends directly on `prog2.c`. The target `prog.exe` then depends indirectly on `prog1.c` and `prog2.c`.

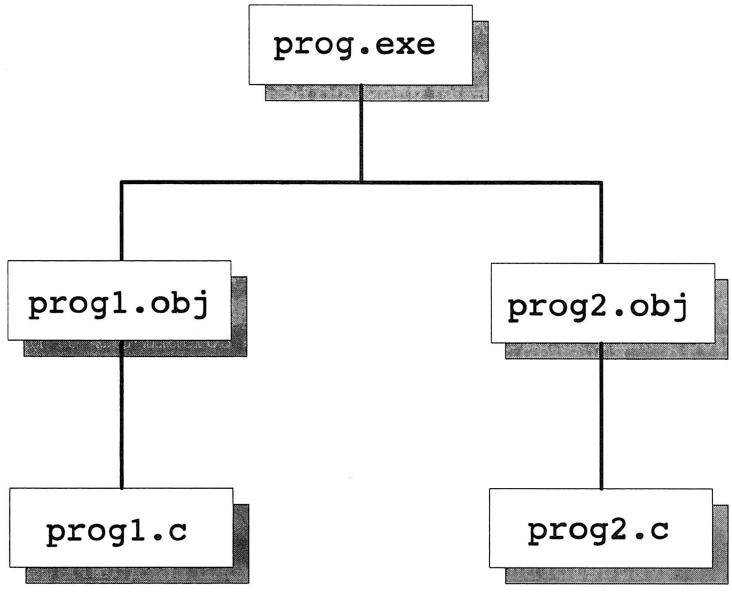


Figure 1.1. Dependency Tree: Indirect Dependencies

MAKE Operations

The usual way to invoke *make* is to issue the command `nmake` at the operating system prompt. (For Borland's MAKE or MAKE on UNIX systems, the command is `make`.) *make* then searches for a makefile in the current directory; the default name of the makefile is `makefile`. If *make* cannot find this file, it issues a message like:

```
Make: No arguments or description file. Stop
```

In our case, *make* finds the makefile. It is our first makefile (from page 2); here it is again:

```
prog.exe:  prog1.obj  prog2.obj
        cl  -o  prog.exe  prog1.obj  prog2.obj

prog1.obj:  prog1.c
        cl  -c  -W2  -O  prog1.c

prog2.obj:  prog2.c
        cl  -c  -W2  -O  prog2.c
```

Because we issued a plain `nmake` command, the target that *make* is going to build is the first target in the makefile; specifically, `prog.exe`. Now *make* sees that `prog1.obj` is a dependency for that target, so it compares the date and time of the object file `prog1.obj` with the date and time of its dependency, `prog1.c`. If `prog1.obj` is older than `prog1.c`, then *make* uses the makefile command

```
cl  -c  -W2  -O  prog1.c
```

to invoke the C compiler and to produce a new `prog1.obj`. On the other hand, if `prog1.obj` is newer than `prog1.c`, then *make* understands that `prog1.obj` was created after the last time `prog1.c` was modified. In this case, *make* does not issue any command because `prog1.c` does not have to be recompiled.

make then checks the date and time of `prog2.obj` and `prog2.c` to determine if `prog2.c` needs to be recompiled.

When all the object files (`.obj` files) are up-to-date with respect to their source files, *make* examines the date and time of `prog.exe` and the date and time of each `.obj` file that we told *make* is necessary to produce the executable file (`.exe` file). In our example, *make* knows which object files to look at from the first line of the makefile:

```
prog.exe:  prog1.obj  prog2.obj
```

If any `.obj` file is newer than the `.exe` file, or if `prog.exe` does not exist yet, then *make* issues the command

```
cl  -o  prog.exe  prog1.obj  prog2.obj
```

to link `prog1.obj`, `prog2.obj`, and the necessary functions from the default C library to produce `prog.exe`. (The compiler command we use — in this case `cl` — determines which C library will be linked in with our object files.) If all the `.obj` files are older than the `.exe` file, then *make* does nothing; it has completed its work.

If we use Borland's C compiler, the name of the compiler command we use in the makefile is different. To invoke the compiler, we use `bcc` (instead of `cl`). This means our makefile would look like this:

```
prog.exe:  prog1.obj  prog2.obj
        bcc  -eprog.exe  prog1.obj  prog2.obj

prog1.obj:  prog1.c
        bcc  -c  -w  -O2  prog1.c

prog2.obj:  prog2.c
        bcc  -c  -w  -O2   prog2.c
```

Summary

make is a utility that simplifies the problem of keeping track of modified files and recompiling and relinking those files to produce an executable program. *make* works by following rules that we provide in a makefile.

make requires us to follow certain rules when we write a makefile. The basic rules are:

1. Start a target/dependency line in the first column of the line.

2. Indent a command line, using one or more blank spaces or tabs. (The PC versions of *make* accept either means of indenting the line; the UNIX version requires a tab.)

3. Use a backslash (\) at the end of a line that is to be carried over to the next line.

4. Use a pound sign (#) to tell *make* that what follows is a comment. A comment can begin anywhere on a line. We can write any number of comment lines, each beginning with a pound sign; however, we cannot use the backslash to indicate the continuation of a comment.

Two basic elements of a makefile are targets and dependencies. *make* works by comparing the time stamp of each target and its dependencies. It recompiles a source file only if the source file is newer than its associated object file.

We can tailor the operation of *make* by specifying a target in the makefile when we issue the `nmake` command. Here are two examples of `nmake` commands:

```
nmake
```

runs *make* on a makefile that appears in the current directory.

```
nmake  target
```

runs *make* on `target` only. When we do not specify a target, as in the first example, *make* builds the first target in the makefile.

Chapter 2

Invoking MAKE

make is invoked interactively from the command line of the operating system. For example, if we are using DOS or OS/2 and the prompt is

```
C>
```

we invoke Microsoft's or IBM's *make* by typing `nmake` after the prompt:

```
C>  nmake
```

We invoke Borland's *make* by typing `make` after the prompt:

```
C>  make
```

We also use the latter command to invoke MAKE on UNIX systems. (In future examples we will show just the `nmake` command, with the understanding that we use `make` for Borland's *make* on DOS or OS/2 or for MAKE on UNIX systems.)

In all cases, the *make* utility searches the current directory for a file called `makefile`.

In Chapter 1, we saw how *make* works when we issue a plain `nmake` command (as we just did), and we saw the effect of invoking *make* with a target. We can control other details of the operation of *make* by issuing various command line options with the *make* command.

Common MAKE Options

We can invoke *make* with many different options. Although the actual options and what they do vary among the different versions of *make*, they

all have some characteristics in common. (In Chapters 5, 6, and 7 we will tell you about the options that apply specifically to DOS, OS/2, and UNIX versions of *make*.) In all versions, an option is represented by a single character (sometimes called a flag) preceded by either a dash (-) or a forward slash (/). UNIX and PC versions use the dash; the forward slash is used only in PC versions.

Here are some of the options we can use with *make*:

`f` *filename*	Use *filename* instead of the default name of `makefile`.
`n`	Go through the makefile without executing any of the commands. This option is helpful in debugging a makefile.
`p`	Print all macros and target information.

We can use as many options as we like with the *make* command, and we can invoke *make* with options and a target. The target must be the last item on the command line.

Refer to Chapters 5, 6, and 7 for a list of other options available for NMAKE by Microsoft, NMAKE by IBM, MAKE by Borland, and UNIX MAKE.

Examples

In Chapter 1 we created a makefile and called it `makefile`. We chose to name our file `makefile` because *make* automatically searches for `makefile`. But we can direct *make* to search for a makefile that has any name. We do that by using the `f` option followed by a file name. For example, assume we rename our makefile to be `prog1.mak`. Now we have to tell *make* that our makefile is `prog1.mak`:

```
nmake  -f  prog1.mak
```

If we want *make* just to tell us what must be done to create the first target in the makefile, without actually performing the task(s), we use the `n` option:

```
nmake  -n  -f  prog1.mak
```

make searches for `makefile`, checks the date and time of targets and dependencies, and reports the necessary actions. To do this, *make* displays each line it would have to execute to build the target. For example, using the makefile that appears on page 2 and again on page 14, when the `.obj` and `.exe` files have not yet been generated, NMAKE produces

```
cl  -c  -W2  -O  prog1.c
cl  -c  -W2  -O  prog2.c
cl  -o  prog.exe  prog1  prog2
```

What if the makefile is not `makefile`, but `prog1.mak`? We then use both the `n` option and the `f` option:

```
nmake  -n  -f  prog1.mak
```

or

```
nmake  -f  prog1.mak  -n
```

As you can see, the options in *make* can appear in any order. Notice one important restriction: when we use the `f` option, the name of the makefile must immediately follow the flag.

Using Hard-Coded Commands

We introduced makefiles in Chapter 1. Recall that a makefile contains the rules that *make* uses to build an executable object. In addition to targets and dependencies, the rules in a makefile include commands. We are not limited to one command per target; we can use as many commands as we want. And we can write the commands in a makefile in one of two ways: we can hard-code them, or we can use macros.

The makefile we used in Chapter 1 had information about what to compile and what to link to produce the target `prog.exe`. Here is that makefile again:

```
prog.exe:  prog1.obj  prog2.obj
        cl  -o  prog.exe  prog1.obj  prog2.obj

prog1.obj:  prog1.c
        cl  -c  -W2  -O  prog1.c

prog2.obj:  prog2.c
        cl  -c  -W2  -O  prog2.c
```

In this makefile, `prog.exe` is the target and `prog1.obj` and `prog2.obj` are the dependencies. This means that, `prog.exe` depends on `prog1.obj` and `prog2.obj`.

Notice that each dependency item has a command associated with it. In these lines

```
prog1.obj:  prog1.c
        cl  -c  -W2  -O  prog1.c
```

`prog1.c` is the dependency and the command associated with it is:

```
cl  -c  -W2  -O  prog1.c
```

This is an example of a hard-coded command: it is an explicit, literal command that appears with the target `prog1.obj` (which is a dependency for `prog.exe`).

We can hard-code as many commands for the target as we want. For example, we can echo a message before compiling the file `prog1.c`:

```
prog1.obj:  prog1.c
        echo  Compiling  prog1.c
        cl  -c  -W2  -O  prog1.c
```

Using Commands with Macros

Hard-coded commands are fixed. This means that, because we used `cl` in `prog1.obj`, *make* will always invoke `cl` whenever it needs to recompile `prog1.c`. However, what happens if we want to use a different compiler?

Remember what we did in Chapter 1 to switch from Microsoft C to Borland's C? We modified the makefile; specifically, we changed `cl` to `bcc`.

We had to repeat the modification in several places. An alternative to making the same change numerous times is to use a macro.

A macro is a makefile variable. We name the macro and assign a value to it. When *make* reads and executes the makefile, the macro is expanded to its assigned value. For example, we used a hard-coded command in `prog1.obj`:

```
prog1.obj:  prog1.c
        cl  -c  -W2  -O  prog1.c
```

As an alternative, we can define a macro to indicate the compiler we want to use:

```
CC = cl
```

Macros usually appear in uppercase so it is easier to discern a macro from a command.

We use `$( )` around the macro name when we want *make* to use the value assigned to the macro:

```
prog1.obj:  prog1.c
        $(CC)  -c  -W2  -O  prog1.c
```

At execution time, *make* interprets `$(CC)` as "the value assigned to `CC`." In this case, the value is `cl`.

We can also define a macro for the compiler options:

```
CFLAGS  =  -c  -W2  -O
```

Using both macros, the command for the target `prog1.obj` then becomes:

```
prog1.obj:  prog1.c
        $(CC)  $(CFLAGS)  prog1.c
```

Now the makefile looks like this:

```
CC      =  cl
CFLAGS  =  -c  -W2  -O

prog.exe:  prog1.obj  prog2.obj
        $(CC)  -o prog.exe  prog1.obj  prog2.obj
```

```
prog1.obj:  prog1.c
        $(CC)  $(CFLAGS)  prog1.c

prog2.obj:  prog2.c
        $(CC)  $(CFLAGS)  prog2.c
```

What if we choose to use Borland's C compiler instead of Microsoft's C compiler? When we use macros consistently in a makefile we have to change only the macros:

```
CC     =  cl
CFLAGS =  -c  -W2  -O
```

The macros become:

```
CC     =  bcc
CFLAGS =  -c  -w  -O2
```

What if we choose to use the C compiler in a UNIX system? We then change the macros to read

```
CC     =  cc
CFLAGS =  -c  -O
```

In this case, we also must change the `.obj` extensions to `.o`. The rest of the makefile remains the same.

Using macros in a makefile offers several advantages over using hard-coded commands. As we've just seen, macros add flexibility to a makefile because they make it easier to modify the commands. In addition, macros help reduce the possibility for introducing errors: the compiler name and the compiler options appear in one place; if they are not right, we need to correct the error in one place only. *make* uses the identical value for a macro every place we refer to the macro.

A macro can contain any text. Sometimes the macro replacement value is a command, such as `cl` or `bcc`; sometimes it is a string that contains a message that we want to display during execution. For example, in the following makefile the macros `COMMENT1`, `COMMENT2`, and `COMMENT3` contain strings that tell us what *make* is doing:

```
CC       =  cl
CFLAGS   =  -c -W2 -O
```

```
COMMENT1 = "Creating prog1.obj file"
COMMENT2 = "Creating prog2.obj file"
COMMENT3 = "Linking prog1.obj and prog2.obj"

prog.exe: prog1.obj prog2.obj
        echo $(COMMENT3)
        $(CC) -o prog.exe prog1.obj prog2.obj

prog1.obj: prog1.c
        echo $(COMMENT1)
        $(CC) $(CFLAGS) prog1.c

prog2.obj: prog2.c
        echo $(COMMENT2)
        $(CC) $(CFLAGS) prog2.c
```

As *make* builds each target, we see the following messages:

```
Creating prog1.obj file
Creating prog2.obj file
Linking prog1.obj and prog2.obj
```

More About Dependencies

A C program, such as `prog1.c` in our example, usually uses `#include` to include a file that contains `#define` statements, function prototypes, or C constants that the program requires. In C, these files are called *include files* or *header files* and usually have a `.h` extension.

What if `prog1.c` contained the line

```
#include   "prog1.h"
```

to include the contents of the file `prog1.h`? We should state that dependency for the target `prog1.obj` in our makefile:

```
prog1.obj: prog1.c prog1.h
        $(CC) $(CFLAGS) prog1.c
```

This means that `prog1.obj` depends on `prog1.c` and `prog1.h`. If either dependency — `prog1.c` or `prog1.h` — has a date or time newer than the date or time of `prog1.obj`, or if `prog1.obj` does not exist yet, then *make* executes the command that follows the target to build `prog1.obj`:

```
$(CC)  $(CFLAGS)  prog1.c
```

The dependency tree for `prog1.obj` appears in Figure 2.1, and the

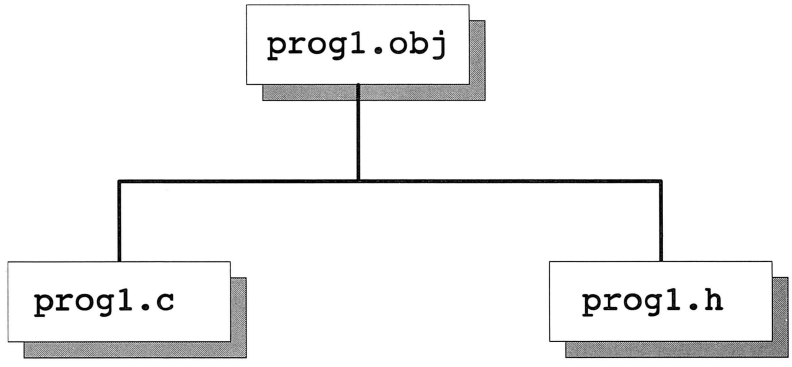


Figure 2.1. Dependency Tree: Header File as a Direct Dependency

dependency tree for `prog.exe` appears in Figure 2.2.

You can write a program that automatically builds dependencies for a makefile. We have written one that builds dependencies for a makefile under DOS and OS/2. The program is called **depends** and we describe it in Chapters 5 and 6 and Appendix I.

Summary

We can control the execution of the *make* command by invoking `nmake` (or `make`) with options. We can also specify a target for *make* to build. For example:

```
nmake  -n
```

or

```
nmake  -n  target
```

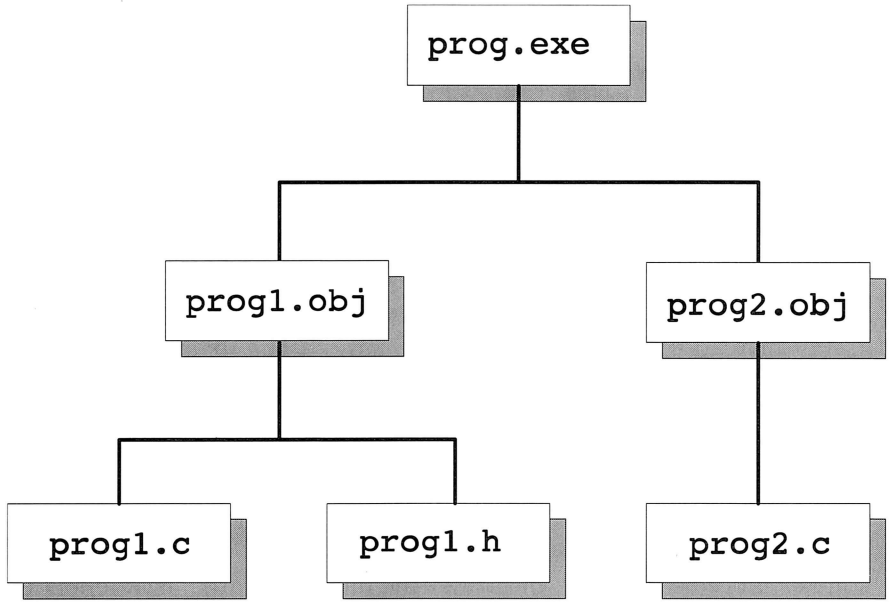


Figure 2.2. Dependency Tree: Header File as an Indirect Dependency

runs *make* without actually building any target. *make* lists the files that will be affected and the commands that will be executed when we run *make* without the `-n` flag. As another example,

```
nmake -f filename
```

or

```
nmake -f filename target
```

runs *make* using the set of rules in `filename` (instead of the default `makefile`). When we use the `f` option, the file name must immediately follow the flag.

The `nmake` command can take any number of options, and the options can appear in any order. When we invoke *make* with a target as well as options, the target must be the last item on the command line.

Each dependency item in a makefile has one or more commands associated with it. The commands can be hard-coded, or they can be written with macros. A hard-coded command appears with its target and is fixed;

for example, the same compiler is used with the same compiler options every time *make* is invoked for the makefile.

Macros provide an alternative to hard-coded commands. We can define any number of macros in a makefile by assigning a value (such as a command or a text string) to each macro name. On a command line, a macro name has `$( )` around it, which tells *make* to use the value assigned to the macro. Macros offer several advantages:

- The makefile is more flexible. We can use a different compiler or specify different compiler options simply by redefining a macro in one place instead of making changes throughout the makefile.
- The makefile is less likely to contain errors. We assign a value to a macro in one place; if we make a mistake, we have to correct it in only one place.

A C program that appears as a dependency in a makefile may include one or more header files. A header file must be stated as a dependency along with the C program in which it is included.

Chapter 3

More About MAKE

In this chapter we continue our progress toward mastering *make*. We revisit and expand on topics introduced in the first two chapters and introduce some new concepts that we will build on when we meet the advanced topics covered in later chapters.

For example, in Chapter 2 we began to learn about the use of macros in makefiles. We also met some of the options we can use when we invoke *make* from the command line. In this chapter, we build on that foundation, learning more about the intricacies of macros and discovering how to find out about the values of macros in our makefile by issuing an option with the `nmake` (or `make`) command.

Among the topics covered in Chapter 1 were targets and explicit commands in makefiles. Pseudotargets are an extension of the general concept of targets; we meet pseudotargets in this chapter and will see them again later in the book. In addition, we will introduce you to inference rules and make good on our promise to tell you about implicit commands.

Finally, this chapter offers two sample makefiles of interest. One serves as a progress check to let you see how far you've come toward mastering *make*. The second illustrates an application of *make* in a real-life project; specifically, the creation of this book.

More About Macros

We introduced macros in Chapter 2. We saw that we can include several macros in a makefile to provide flexibility and consistency and to eliminate

the possibility of errors.

It is a common practice (although not required) to use capital letters for macro names. Be aware, however, that all versions of *make* are case sensitive and see `CFLAGS` and `Cflags`, for example, as two different macro names.

Another point of style is to declare all our macros at the beginning of the makefile. This technique makes it easier to find and modify macros as the project develops.

The macros we added to our makefile in Chapter 2 are called user-defined macros; that is, they are macros we created to use in our makefile. The term *user-defined* distinguishes these macros from another type of macro — *predefined* (or built-in) macros. A predefined macro is a macro whose definition is available every time we start *make*.

Let's review our makefile from Chapter 2, then make some changes to illustrate the use of user-defined and predefined macros. Here is the makefile from Chapter 2:

```
CC      =  cl
CFLAGS  =  -c  -W2  -O

prog.exe:  prog1.obj  prog2.obj
        $(CC)  -o prog.exe  prog1.obj  prog2.obj

prog1.obj:  prog1.c
        $(CC)  $(CFLAGS)  prog1.c

prog2.obj:  prog2.c
        $(CC)  $(CFLAGS)  prog2.c
```

User-Defined Macros

An example of a user-defined macro is `CFLAGS`, which we used in the makefile in Chapter 2:

```
CFLAGS =  -c  -W2  -O
```

User-defined macros serve a number of purposes; for example, to refer easily and consistently to a utility (such as a C compiler) and to the flags that we want to apply to that utility (as we defined in `CFLAGS`).

Another common use for user-defined macros is to identify the object files we need to create to build the executable program. In our example, we want to build `prog.exe`, and the object files we need are `prog1.obj` and `prog2.obj`. To make it easy to refer to these object files, we define a macro to be a synonym for them:

```
OBJS  =  prog1.obj  prog2.obj
```

We define another macro to identify the target we want to build:

```
TARGET  =  prog.exe
```

Using these new user-defined macros along with the macros we defined in Chapter 2, we can change the makefile to read:

```
CC     =  cl
CFLAGS =  -c  -W2  -O
OBJS   =  prog1.obj  prog2.obj
TARGET =  prog.exe

$(TARGET):  $(OBJS)
        $(CC)  -o  $(TARGET)  $(OBJS)

prog1.obj:  prog1.c
        $(CC)  $(CFLAGS)  prog1.c

prog2.obj:  prog2.c
        $(CC)  $(CFLAGS)  prog2.c
```

Predefined Macros

A predefined macro is a macro that the *make* utility already knows. In other words, it is a macro whose definition is available every time we start *make*.

Predefined macros vary with the version of *make* we are using. We will talk about some common predefined macros in this section and reserve for Chapters 5, 6, and 7 the discussion of specific macros available with each version of *make*.

Our first predefined macro is `CC`, which identifies the C compiler on our system. In Microsoft's NMAKE the predefined macro `CC` is equivalent to:

```
CC = cl
```

In IBM's NMAKE the predefined macro CC is equivalent to:

```
CC = icc
```

In Borland's MAKE on both DOS and OS/2 systems the predefined macro CC is equivalent to:

```
CC = bcc
```

In the UNIX version of MAKE the predefined macro CC is equivalent to:

```
CC = cc
```

Well, now — here's a pleasant surprise. The CC macro is predefined. This means we do not have to explicitly define CC in our makefile.

Another predefined macro is $@. This macro is a synonym for the current target. In this fragment from our makefile

```
prog.exe:  prog1.obj  prog2.obj
        $(CC)  -o  prog.exe  prog1.obj  prog2.obj
```

the target is prog.exe, so the value of $@ is prog.exe. Notice that on the command line we ask CC to put its output in the file prog.exe:

```
-o  prog.exe
```

Instead of retyping the target name, we can use the predefined macro $@ because it is a synonym for the current target:

```
prog.exe:  prog1.obj  prog2.obj
        $(CC)  -o  $@  prog1.obj  prog2.obj
```

Our next predefined macro is $*. This macro is a synonym for the target name *without* an extension. $* includes the path information. Consider this fragment, for example:

```
prog1.obj:  prog1.c
        $(CC)  $(CFLAGS)  prog1.c
```

Here the target is prog1.obj. The target without an extension is prog1, so the value of $* is prog1. Using this predefined macro, we can change the command line for the target prog1.obj to read

```
prog1.obj:  prog1.c
        $(CC)  $(CFLAGS)  $*.c
```

which still means "apply the value of CC with the value of CFLAGS to prog1.c."

If we use the user-defined macro CFLAGS (from Chapter 2) and the three predefined macros CC, $@, and $*, our makefile becomes:

```
CFLAGS  =  -c -W2  -O

prog.exe:  prog1.obj  prog2.obj
        $(CC)  -o  $@  prog1.obj  prog2.obj

prog1.obj:  prog1.c
        $(CC)  $(CFLAGS)  $*.c

prog2.obj:  prog2.c
        $(CC)  $(CFLAGS)  $*.c
```

Notice that we have removed the macro definition

```
CC = cl
```

from the makefile because it is a predefined macro for the version of *make* we are using.

On the first command line, we have replaced the target file name prog.exe with $@; and on the other command lines, we have replaced the source file names prog1.c and prog2.c with $*.c. The macro $@ takes the value of the current target; and the macro $* takes the value of the current target without an extension.

Now let's add the macros we defined at the beginning of this chapter. We can see that our makefile has changed considerably from where we started:

```
CFLAGS  =  -c  -W2  -O
OBJS    =  prog1.obj  prog2.obj
TARGET  =  prog.exe

$(TARGET):  $(OBJS)
        $(CC)  -o  $@  $(OBJS)

prog1.obj:  prog1.c
        $(CC)  $(CFLAGS)  $*.c
```

```
prog2.obj:  prog2.c
        $(CC)  $(CFLAGS)  $*.c
```

In this example, the user-defined macro `TARGET` expands to `prog.exe`, so the value of the predefined macro `$@` is `prog.exe`.

Another useful predefined macro is `$<`. We discuss `$<` on page 30.

You may have noticed in this discussion of predefined macros that we have not used parentheses around them: we use `$@` instead of `$(@)`, `$*` instead of `$(*)`, and so on. The parentheses are not needed in these cases because the macro name consists of a single character. Parentheses are required only when the macro name consists of more than one character.

(Every rule has its exception. The exception to this rule is the predefined macro `$**`, which you will find described in the documentation that came with your compiler.)

References to single-character user-defined macros can also omit the parentheses. For example, we could define the macro name `T` to represent the target in our makefile. In this case, we would use `$T` to refer to `prog.exe`. In general, however, single-character macro names are less useful, because they are less meaningful, than longer macro names.

Macros and the p Option

Sometimes we need to verify the values of macros that we are using in a makefile. We may want to check the default values that *make* is using, or we think the default values are not what we want to use. An easy way to get this information is to invoke *make* with the `n` option and the `p` option:

```
nmake  -n  -p
```

Recall from Chapter 2 that the `n` option tells *make* to check what it should execute in a makefile without actually executing anything. The `p` option tells *make* to display the macro information. This is an easy way to find out the values of the macros *make* is using in the makefile.

Pseudotargets

In Chapter 1 we defined a target to be a name that appears at the beginning of a line and is followed by a colon (`:`). An important exception to this general definition is the pseudotarget.

Pseudotargets look like targets. However, unlike true targets, they are not files. This fact elicits a word of caution: no file, whether currently existing or created in the future, may share the name of a pseudotarget in the same makefile.

A pseudotarget can be user-defined or predefined. The next two sections distinguish between the two.

User-Defined Pseudotargets

User-defined pseudotargets typically appear toward the end of the makefile (although they certainly are not restricted to that placement). A user-defined pseudotarget is a place holder for one or more commands. Occasionally, we need to execute those commands, so we create a pseudotarget and list the commands after it. For example, in these lines

```
cleanup:
        erase *.obj     # erase object files
        erase *.exe     # erase executable files
```

`cleanup` is a user-defined pseudotarget. Two `erase` commands are associated with it. When we issue

```
nmake  cleanup
```

on the command line, *make* executes the commands following the pseudotarget. `cleanup` is not a file and it does not have dependencies (there are no file names after the colon (`:`)).

A pseudotarget without dependencies is always true; that is, *make* will invoke all the commands associated with the pseudotarget every time *make* either reaches the pseudotarget (via a dependency) or is invoked with the name of the pseudotarget.

Predefined Pseudotargets

Predefined pseudotargets are names that begin with a dot (`.`) and sometimes are called *dot directives*. (Borland, in particular, uses this terminology.) Except for the two versions of MAKE from Borland, which support lowercase pseudotarget names, the name of a predefined pseudotarget consists of uppercase letters. As with true targets and user-defined pseudotargets, the name of the predefined pseudotarget is followed by a colon. For example,

```
.SILENT:
```

is a predefined pseudotarget. Predefined pseudotargets usually appear at the beginning of the makefile.

`.SILENT:` tells *make* to not display lines as they are executed. This is equivalent to invoking *make* with the `s` option:

```
nmake  -s
```

Another pseudotarget is

```
.IGNORE:
```

which tells *make* to ignore the codes returned by the commands that *make* invokes to accomplish its job. This is equivalent to using the `i` option with *make*.

Why use the `.IGNORE:` pseudotarget? Regardless of the version of *make* we use, *make* abandons execution any time a command it has invoked fails. In this case, *make* receives a return code that indicates an error. Sometimes we do not want *make* to stop if an error is encountered. `.IGNORE:` tells *make* to ignore the return codes and to continue executing.

By now you will recognize `.SUFFIXES:` to be another predefined pseudotarget. All versions of *make* covered in this book support `.SUFFIXES:` (although other versions may not). The versions of *make* that support `.SUFFIXES:` (or, in Borland's case, `.suffixes:`) maintain a built-in list of extensions and inference rules. (Although inference rules are closely tied to the use of this pseudotarget, we will wait until the next section to discuss them in detail.)

We can add to the extensions that *make* knows. When we use `.SUFFIXES:` with one or more extensions, *make* adds those extensions to its internal list. For example,

```
.SUFFIXES:  .inc  .tex
```

adds the file extensions `.inc` and `.tex` to the list of extensions that *make* already knows how to handle.

When *make* finds a target without dependencies, *make* searches for a file that has the same name as the target, with an extension from the list of known extensions. For example,

```
.SUFFIXES:  .inc  .tex

      ⋮

xyz.obj:
        command
```

tells *make* to search for `xyz` with one of the extensions from the built-in list (such as `.c`, for example). If the search fails, *make* looks for a file named `xyz` with one of the extensions associated with the `.SUFFIXES:` pseudotarget in the makefile, in the order in which the extensions are listed.

In the case of our example, after exhausting the built-in list, MAKE searches `xyz.inc`; if *make* finds it, that is the file that *make* will use and the search ends. However, if `xyz.inc` does not exist, *make* searches for `xyz.tex`.

Sometimes we may choose to tell *make* to ignore its built-in list. To do this, we use `.SUFFIXES:` without any file extension following it. If we then use another `.SUFFIXES:` with extensions, we guarantee that *make* knows only those extensions. For example,

```
.SUFFIXES:
.SUFFIXES:  .inc  .tex
```

first clears the list of file extensions that *make* knows initially, then adds `.inc` and `.tex` to the now-empty list.

Inference Rules and Implicit Commands

An inference rule provides *make* with a general rule that it can use to build a target with a given extension from a dependency with a given extension. Because the rule applies to the extension of a file name, it eliminates the need to repeat a given command in a makefile. Note that a file extension also is called a suffix; for this reason, you may hear inference rules referred to as *suffix rules.*

In our example makefile, we've seen several file name extensions (or suffixes): `exe`, `obj`, `c`. Just as with an extension to a file name, each extension in an inference rule is preceded by a dot. Thus, an inference rule always begins with a dot (`.`), which is followed by an extension, then another dot, and finally another extension. For example,

```
.c.obj:
        $(CC)  $(CFLAGS)  $*.c
```

is an inference rule. This rule starts with `.c.obj` (that is, the two extensions `c` and `obj`, each preceded by a dot). The rule indicates that we can have a `.c` file as our source and can generate a `.obj` file from it. The command line

```
$(CC)  $(CFLAGS)  $*.c
```

indicates how *make* will accomplish that:

- Use the value of `CC` to invoke the C compiler with the flags specified in `CFLAGS`.
- Use the current target without an extension (`$*`).
- Add the extension (suffix) `c` to `$*` to build the file name that `CC` should compile.

An inference rule, then, consists of an implicit target, an implicit dependency, and one or more implicit commands. In this case, the implicit target is the `.obj` file to be generated; the implicit dependency is the `.c` source file to be compiled to create the target; and the implicit command is:

```
$(CC)  $(CFLAGS)  $*.c
```

When a makefile contains a target and a list of dependencies without an explicit command line, *make* searches for an inference rule it can apply. When it finds the applicable inference rule, *make* executes the associated implicit commands. The commands are implicit — as contrasted with the explicit commands we talked about in Chapter 1 — because they do not appear explicitly with the target and dependencies to which they are applied.

It is time to discuss another predefined macro: `$<`. This macro is a synonym for the dependent file that is out-of-date with respect to the target. For example, in the inference rule we discussed earlier,

```
.c.obj:
        $(CC)  $(CFLAGS)  $*.c
```

`$*` is the current target without an extension. `$*.c` is then the name of the file to be compiled. Because `$<` is a synonym for the dependent file in inference rules, we can use

```
.c.obj:
        $(CC)  $(CFLAGS)  $<
```

to convey the same information as the earlier rule. However, either form (`$*.c` or `$<`) is valid.

Most versions of *make* allow `$<` to appear only in inference rules.

In the next section, we will see some examples of how to use inference rules and implicit commands. As with macros and pseudotargets, there are two types of inference rules: user-defined and predefined. We will use our makefile from Chapter 2 to show how both types of inference rules are used.

User-Defined Inference Rules

We learned earlier in this chapter that `CC` is a predefined macro, so we have removed its definition from the makefile. With that change, here is the makefile from Chapter 2 again:

```
CFLAGS  =  -c  -W2  -O

prog.exe:  prog1.obj  prog2.obj
        $(CC)  -o  prog.exe  prog1.obj  prog2.obj

prog1.obj:  prog1.c
        $(CC)  $(CFLAGS)  prog1.c

prog2.obj:  prog2.c
        $(CC)  $(CFLAGS)  prog2.c
```

We have two macros: the predefined macro `CC` and the user-defined macro `CFLAGS`. Every time we invoke the compiler, we use one or both of these macros. The macro mechanism provides uniformity in the makefile. For example, we initialize `CFLAGS` in one place, then use the value of `CFLAGS` — `$(CFLAGS)` — wherever we refer to the C compiler. But we still explicitly invoke the compiler when we want to convert a `.c` file into a `.obj` file:

```
prog1.obj:  prog1.c
        $(CC)  $(CFLAGS)  prog1.c
```

As we said earlier, the inference rule provides a general rule for creating a new file from an existing file. Recall, for example, that to convert a `.c` file into a `.obj` file we can use the inference rule:

```
.c.obj:
        $(CC)  $(CFLAGS)  $*.c
```

This is an inference rule we created for *make* to use when it determines that a `.c` file needs to be compiled (in effect, *make* generates a `.obj` file from a `.c` file).

Remember that when the makefile has a target and a list of dependencies without an explicit command line, *make* searches for an inference rule that it can apply. For example, using the inference rule

```
.c.obj:
        $(CC)  $(CFLAGS)  $*.c
```

and the target/dependency line

```
prog1.obj:  prog1.c
```

make applies the command line

```
$(CC)  $(CFLAGS)  $*.c
```

to build `prog1.obj` from `prog1.c`. Remember, we defined:

```
CFLAGS  =  -c  -W2  -O
```

In our version of *make*, `CC` is predefined:

```
CC  =  cl
```

And `$*` is predefined to be the name of the current target without an extension. Then *make* understands that the command line, in this case, is:

```
cl  -c  -W2  -O  prog1.c
```

Using macros and an inference rule, the makefile becomes:

```
CFLAGS  =  -c  -W2  -O

.c.obj:
        $(CC)  $(CFLAGS)  $*.c
```

```
prog.exe:  prog1.obj  prog2.obj
        $(CC)  -o  prog.exe  prog1.obj  prog2.obj

prog1.obj:  prog1.c

prog2.obj:  prog2.c
```

Notice that the command lines we saw earlier are no longer present after the targets `prog1.obj` and `prog2.obj`; that is, these targets do not have any explicit commands associated with them. *make* knows *implicitly* how to build `prog1.obj` from `prog1.c` and `prog2.obj` from `prog2.c` by applying the command line from the inference rule `.c.obj`.

We can use the predefined macro `$@`, define our own macros `OBJS` and `TARGET` (as we did earlier in this chapter), and construct a new makefile that is easier to maintain:

```
CFLAGS =  -c  -W2  -O
OBJS   =  prog1.obj  prog2.obj
TARGET =  prog.exe

.c.obj:
        $(CC)  $(CFLAGS)  $*.c

$(TARGET):  $(OBJS)
        $(CC)  -o  $@  $(OBJS)
```

We do not need to include the targets `prog1.obj:` and `prog2.obj:` because *make* knows how to find the dependencies based on the inference rules.

We can use any number of commands in an inference rule. In this example, one command was sufficient. We can create inference rules for any conversion we find necessary in our makefile. For example, when we deal with assembly language, we define the macros

```
AS     =  masm
AFLAGS =
```

because we want to use Microsoft's `masm` assembler without any flags. The inference rule we want to use in this case is:

```
.asm.obj:
        $(AS)  $(AFLAGS)  $*.asm  ;
```

Why have we defined AFLAGS to be empty? Simply put, at this time we do not need any options. AFLAGS is present as a place holder. If we decide later that we need one or more options to apply to the assembler, we change AFLAGS without disturbing the rest of the makefile.

The technique of defining an empty macro is an example of planning ahead in a development project. A more experienced software developer might define an empty macro almost without thinking about it, while a less experienced programmer might completely overlook the usefulness of such a place holder.

Predefined Inference Rules

Most versions of *make* know about common file extensions, such as .c and .obj. This means that, in many instances, *make* already has a predefined rule for obtaining a .obj file from a .c file.

For example, NMAKE from Microsoft uses the predefined rule

```
.c.obj:
        $(CC)  $(CFLAGS)  /c  $*.c
```

where the macros CC and CFLAGS in NMAKE are preset to:

```
CC     =  cl
CFLAGS =
```

NMAKE from IBM uses the predefined rule

```
.c.obj:
        $(CC)  $(CFLAGS)  /c  $*.c
```

where the macros CC and CFLAGS in NMAKE are preset to:

```
CC     =  icc
CFLAGS =
```

MAKE from Borland uses the predefined rule

```
.c.obj:
        $(CC)  $(CFLAGS)  /c  $&.c
```

where the macros CC and CFLAGS in MAKE are preset to:

```
CC     = bcc
CFLAGS =
```

Notice that Borland's predefined inference rule uses `$&.c` instead of `$*.c`. Remember, `$*` includes path information; in contrast, `$&` does not. MAKE on UNIX systems uses the predefined rule

```
.c.o:
        $(CC)  $(CFLAGS)  -c  $<
```

where the macros `CC` and `CFLAGS` in MAKE are preset to:

```
CC     = cc
CFLAGS = -O
```

As we saw in the previous section, a macro can be empty. In three of the preceding cases, `CFLAGS` is an empty macro.

Now, using our own macros as defined earlier and the predefined inference rules of NMAKE, we can change our makefile to read:

```
CFLAGS = -W2  -O
OBJS   = prog1.obj  prog2.obj
TARGET = prog.exe

$(TARGET):  $(OBJS)
        $(CC)  -o  $@  $(OBJS)
```

For another example, let's modify our makefile to run on a UNIX system. Before we do that, there are some things we need to know about files in UNIX systems. For example, in UNIX systems (as in DOS and OS/2) the extension for a C file is `.c`. However, the extension for an object file in UNIX systems is `.o` (unlike DOS and OS/2, where it is `.obj`). Here's something else we should know about UNIX files: executable files are not required to have special extensions like `.exe` or `.com`. So we'll call our executable file `prog`.

We do not need to define an inference rule to produce a `.o` file from a `.c` file because that inference rule is predefined. Nor do we need to initialize the `CC` and `CFLAGS` macros because their predefined values are acceptable for our purposes. The makefile then becomes:

```
OBJS   =  prog1.o  prog2.o
TARGET =  prog

$(TARGET):  $(OBJS)
        $(CC)  -o  $@  $(OBJS)
```

Your version of *make* may support other inference rules. Remember, too, that invoking *make* with the `p` option displays all predefined values, including predefined inference rules.

How Are We Doing?

Let's stop here and see how much we've learned about makefiles. By now, you should be able to read a makefile and understand its elements, the relationships among them, and the functions the makefile performs.

The next example will give you a chance to test yourself on what you've learned. But before we show you the example, let's talk about what we mean by being able to read and understand a makefile.

The purpose of this book is to guide you through the process of creating a makefile that you can apply to a particular program development project. It is not our purpose to teach you about your compiler. So when you see a macro definition like

```
CFLAGS =  -c  -W2  -O
```

and a command line that includes macros like

```
$(CC)  $(CFLAGS)  prog1.c
```

you should understand that *make* expands the macro `$(CFLAGS)` to its value `-c -W2 -O`. If you know, for example, that `-W2` is a compiler option that means "set warnings to level 2," you're in great shape! But if you don't know that fact, it doesn't mean you don't understand makefiles; rather, it means you need to refer to your compiler documentation to find out more about compiler options.

Here is an example makefile for you to test yourself on. If you can read it and understand what *make* will do as it executes the makefile, you are

ready to move on to the advanced topics in the next chapter. If you aren't completely sure of your understanding of this example, you should review the information we have presented up to now.

```
# Define suffixes

.SUFFIXES:
.SUFFIXES:  .c  .asm  .obj  .exe

# Define macros

AFLAGS  =
CFLAGS  =  -c  -W2  -O
OBJS    =  sample1.obj  sample2.obj
TARGET  =  sample

# Define inference rules

.asm.obj:
        $(AS)  $(AFLAGS)  $*.asm ;

.c.obj:
        $(CC)  $(CFLAGS)  $*.c

# Define target

$(TARGET).exe: $(OBJS)
        $(CC)  -o  $@  $(OBJS)
        copy   $@  a:$@
        attrib +r  a:$@

# List indirect dependencies

sample1.obj:    sample1.c              \
                sample1.h

sample2.obj:    sample2.asm            \
                sample.inc
```

You might wonder why we did not use `$*` in the list of indirect dependencies. In common practice, `$*` is used only in inference rules and command lines. Now, here are some hints:

- See "Predefined Pseudotargets" beginning on page 27 for information about `.SUFFIXES:`.
- See "Syntax Rules" beginning on page 2 for the use of `#`.
- See "More About Macros" beginning on page 21 for a discussion of user-defined and predefined macros.
- See "Inference Rules and Implicit Commands" beginning on page 29 for information about inference rules.
- See "Syntax Rules" beginning on page 2 for the use of `\`.

Application: Makefile for Text Formatting

So far our makefiles have dealt with compiling and linking files. However, we can apply *make* to other tasks. For example, we used *make* to assemble this book. We began by creating one file per chapter. We gave each chapter a `.inc` extension, so our chapter files are named `ch1.inc`, `ch2.inc`, and so on.

In addition to creating the text chapters, we wrote a program, which we called `include`. This program copies (includes) examples from our system directly into the text of the chapter. The text for a chapter plus the `included` examples form a `.tex` file. The final product is a device-independent file with the extension `.dvi`.

Before we get into the makefile we created to accomplish the task of assembling this book, let's talk about some terms you'll see:

- `TEXFILES` is a macro we defined in the makefile to represent all the chapters and appendices with examples included.
- `book.tex` is a small file that refers to the individual chapters and appendices that make up the book.
- `latex` is a command that invokes the collection of macros that our text formatting program uses to format our files.

In brief, when `latex` is invoked it produces the `.dvi` file from `book.tex` and `TEXFILES`. Now for the details.

We started our makefile by telling *make* to forget any knowledge of predefined inference rules and suffixes:

```
.SUFFIXES:
```

Then we stated which suffixes are valid for this makefile:

```
.SUFFIXES:      .inc .tex .dvi
```

The inference rule we defined for generating a `.tex` file from a `.inc` file is:

```
.inc.tex:
        include  <$*.inc  >$*.tex
```

In other words, a `.inc` file provides the input to the program `include`, which writes its output to a `.tex` file. Thus, for example, when *make* encounters the target `ch1.tex` in the makefile as it executes, it looks for `ch1.inc` and directs the contents of that file to the program `include`, which in turn creates the file `ch1.tex`.

The target `book.dvi` depends on the value assigned to the user-defined macro `TEXFILES` and `book.tex`:

```
book.dvi:       $(TEXFILES)  book.tex
        latex  book.tex
```

Here is the complete makefile:

```
.SUFFIXES:
.SUFFIXES:      .inc .tex .dvi

TEXFILES=       pre2.tex        \
                pre1.tex        \
                ch1.tex         \
                ch2.tex         \
                ch3.tex         \
                ch4.tex         \
                ch5.tex         \
                ch6.tex         \
                ch7.tex         \
```

```
                apa.tex         \
                apb.tex         \
                apc.tex         \
                apd.tex         \
                ape.tex         \
                apf.tex         \
                apg.tex         \
                aph.tex         \
                api.tex

.inc.tex:
        include  <$*.inc  >$*.tex

book.dvi:       $(TEXFILES)  book.tex
        latex  book.tex
```

Summary

We can use two types of macros in a makefile:

- user-defined
- predefined

As a matter of convention, we capitalize the names of user-defined macros and place them at the beginning of the makefile.

We use `$( )` around the macro name when we want *make* to use the value assigned to the macro. When the macro name consists of one character (such as some of the predefined macros), the parentheses are not required. For example, we use `$(CC)` (more than one character), but `$*` (one character). The only exception to this rule is the predefined macro `$**` (refer to Appendix A).

A special type of user-defined macro is an empty macro, which we create to serve as a place holder for future use as the project develops.

We can find out the names and default values of predefined and user-defined macros, as well as inference rules and pseudotargets, by invoking *make* on the command line with the `p` option and the `n` option:

```
nmake  -p  -n
```

If necessary, we can change the value of any predefined macro by assigning it a new value in the same way that we assign a value to a user-defined macro. By convention, we place redefined macros (along with user-defined macros) at the beginning of the makefile.

Some predefined macros are:

- `CC = cl` (for NMAKE from Microsoft)
- `CC = icc` (for NMAKE from IBM)
- `CC = bcc` (for MAKE from Borland)
- `CC = cc` (for MAKE in UNIX systems)
- `$*` = the current target without an extension (the base file name), with path. For example, in

```
prog1.obj:  prog1.c
            $(CC)  $(CFLAGS)  $*.c
```

  the value of `$*` is `prog1`. `$*` is commonly used only in inference rules and command lines.

- `$@` = the current target (including the extension, if any). For example, in

```
prog.exe:   $(OBJS)
            $(CC)  -o  $@  $*.c
```

  the value of `$@` is `prog.exe`.

- `$&` = the current target without an extension (the base file name), without path (Borland only). For example, in

```
prog1.obj:  prog1.c
            $(CC)  $(CFLAGS)  $&.c
```

  the value of `$&` is `prog1`. Borland uses `$&` in inference rules.

- `$<` = a dependent file out-of-date with the target file. For example,

```
.c.obj:
            $(CC)  $(CFLAGS)  $<
```

  Notice that `$<`, in an inference rule such as `.c.obj`, is equivalent to `$*.c`:

```
.c.obj:
            $(CC)  $(CFLAGS)  $*.c
```

  The internal macro `$<` is used mostly in UNIX systems. `$<` can be used only in inference rules.

A pseudotarget is, as the name implies, not a true target. No file is associated with the target name, and no dependencies follow the target name in the makefile. As with macros, there are two varieties of pseudotargets: user-defined and predefined.

User-defined pseudotargets conventionally appear toward the end of the makefile (in contrast to user-defined macros). The commands associated with the pseudotarget are always executed, either when *make* reaches the pseudotarget in the makefile or when *make* is invoked with the pseudotarget on the command line (similar to invoking *make* with the name of a true target as we did in Chapter 1).

Predefined pseudotargets begin with a dot (and, for that reason, are referred to as dot directives in Borland) and end with a colon. The names of predefined pseudotargets are capitalized, except in Borland's MAKE, which uses lowercase characters. Some predefined pseudotargets are:

`.SILENT:` Do not display command lines from the makefile as they are executed. Equivalent to invoking *make* with the `s` option.

`.IGNORE:` Ignore codes returned by commands that *make* invokes as it executes. Equivalent to invoking *make* with the `i` option.

`.SUFFIXES:` Maintains a built-in list of file extensions that *make* uses to build a target that does not have any dependencies.

A useful application of the `.SUFFIXES:` pseudotarget is to create a list of project-specific suffixes (or file extensions) after first telling *make* to forget the suffixes it already knows about; for example:

```
.SUFFIXES:
.SUFFIXES:  .inc  .tex  .dvi
```

An inference rule (also called a suffix rule) consists of an implicit target, an implicit dependency, and one or more implicit commands. *make* uses inference rules to determine what action to take when it finds a target/dependency line in a makefile without any explicit commands.

The first line of an inference rule contains the extension of a source file (which provides the implicit dependency) followed by the extension of a file to be built (which is the implicit target). A dot precedes each extension, and the line ends with a colon. The implicit commands associated with the inference rule begin on the next line and adhere to the same rules of indentation as explicit commands (refer to Chapter 1).

All versions of *make* support both user-defined and predefined inference rules.

A predefined inference rule for NMAKE from both Microsoft and IBM is:

```
.c.obj:
        $(CC)  $(CFLAGS)  /c  $*.c
```

A predefined inference rule for MAKE from Borland is:

```
.c.obj:
        $(CC)  $(CFLAGS)  /c  $&.c
```

A predefined inference rule for MAKE in UNIX systems is:

```
.c.o:
        $(CC)  $(CFLAGS)  -c  $<
```

Chapters 5, 6, and 7 list more inference rules for these versions of *make*.

Chapter 4

Advanced Topics

As we've seen, *make* is a powerful tool that can perform a wide variety of the tasks associated with managing software development. We've learned that by using macros, pseudotargets, and inference rules we can expand the usefulness of our makefile and, thus, the power of *make*.

We're ready now to begin investigating more advanced ways we can use *make*. Recursion, environment variables, and macro information substitution are techniques available with all versions of *make*. We'll see in this chapter how to take advantage of these powerful devices. Advanced techniques that apply to only one or another specific version of *make* are covered, as appropriate, in the concluding chapters.

Using MAKE Recursively

We can get *make* to do more work for us by using *make* recursively. To put it another way, we can apply the property of recursion to *make*. What does this mean? What is recursion?

In simple terms, for our case recursion means that *make* can reexecute itself, if necessary, and possibly use different makefiles in the process. Let's look briefly at a small example that will help clear up this matter of recursion.

Suppose we have a directory called `top`. Below this directory are three directories called `middle1`, `middle2`, and `middle3`. Figure 4.1 shows the tree for this directory structure.

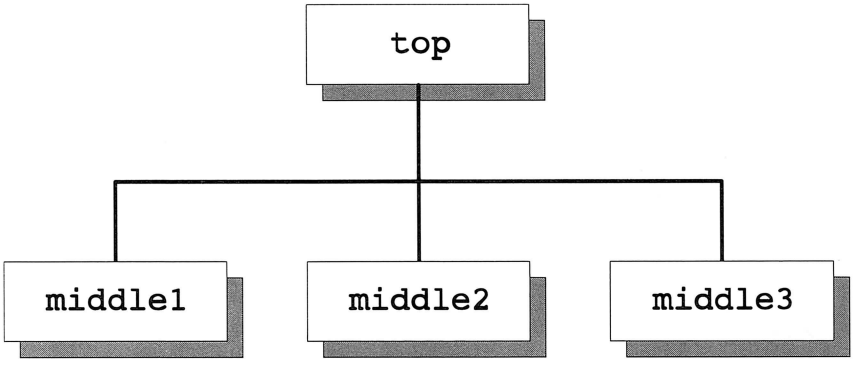


Figure 4.1. Directory Structure

Now, with this structure in mind, let's say each subdirectory contains source code files and a makefile. We want to create a makefile in `top` that will invoke *make* in each of the three subdirectories.

What would that makefile look like? Here is an example of a makefile to perform that very task:

```
all:
        cd  middle1
        $(MAKE)
        cd  ..\middle2
        $(MAKE)
        cd  ..\middle3
        $(MAKE)
        cd  ..
```

The last command (`cd ..`) brings us back to the parent directory of `middle3`; that is, it brings us back to the directory where *make* started executing commands. (Note that this sequence of commands in UNIX systems will not behave the same way as it does in DOS and OS/2. For a UNIX example, see the makefile on page 51 and an explanation in "Makefile Command Lines" on page 194.)

`MAKE` is a predefined macro that invokes the current *make* command. In NMAKE, the macro is preset to

```
MAKE  =  nmake
```

while Borland's macro MAKE is preset to:

```
MAKE = make
```

The makefile above certainly looks very different from the ones we've seen in earlier chapters. Let's look more closely at this makefile.

The first thing we notice is that this makefile has *no* dependencies, and, in place of the usual target, we are using a pseudotarget called `all`. We met pseudotargets in Chapter 3 (page 27). Because the pseudotarget `all` has no dependencies, the original instance of *make* issues all the commands that follow the pseudotarget.

Notice that this makefile invokes *make* three separate times, once for each subdirectory. This gives us a clear picture of how recursion works. We invoked the first instance of *make* from the command line; then *make* invokes itself (the recursion process) by executing the command lines contained in the first makefile.

We have assumed in our example that each subdirectory contains a makefile; specifically, one with the default file name *make* searches for. If our assumption is wrong — if any of the subdirectories lacks this makefile — *make* will display an error message informing us of the problem.

In our first example of using *make* recursively, the makefile expected a separate makefile each time the `nmake` command was issued. Another way to apply recursion is to use the same makefile for each instance of *make*. For example, we may need to reinvoke *make* within our makefile and have it use the same makefile, but with a different target. Here's a makefile that shows this recursive use of *make*:

```
# Recursion using the same makefile (Microsoft)

# Define suffixes

.SUFFIXES:
.SUFFIXES: .c .obj .exe

# Define macros

CFLAGS          = -c  -W2  -O
MID_DIR         = middle1
OBJS            = prog1.obj  prog2.obj
EXEC_NAME       = prog
```

```
# Define target and dependencies

$(EXEC_NAME).exe:  $(OBJS) $(MID_DIR)\prog3.obj
        $(CC) -o $@ $(OBJS) $(MID_DIR)\prog3.obj

$(MID_DIR)\prog3.obj:  $(MID_DIR)\prog3.c
        cd  $(MID_DIR)
        $(MAKE)  -f ..\makefile                    \
                COMM="Compiling PROG3.C"  prog3.obj
        cd ..

prog3.obj:      prog3.c
        echo  $(COMM)
        $(CC) $(CFLAGS) prog3.c
```

Notice that we have added a new direct dependency to `prog.exe`. It is an object file called `prog3.obj`, which resides in the subdirectory `middle1`. Before `prog3.obj` can be created, we must change directories to `middle1` and, while in that subdirectory, we reinvoke *make* to execute the makefile in the parent directory; that is, the same makefile currently being executed by the first instance of *make*.

Let's look again at a couple of sections of our makefile. The section where *make* is reinvoked

```
$(MID_DIR)\prog3.obj:  $(MID_DIR)\prog3.c
        cd  $(MID_DIR)
        $(MAKE)  -f ..\makefile                    \
                COMM="Compiling PROG3.C"  prog3.obj
        cd ..
```

and the indirect dependency line for `prog3.obj`:

```
prog3.obj:      prog3.c
        echo  $(COMM)
        $(CC) $(CFLAGS) prog3.c
```

Notice how we have used the macro we named `MID_DIR` as a path to the files `prog3.obj` and `prog3.c`. We use this technique because the initial instance of `make` is running from `top`, the parent directory, so we have to tell *make* where to find `prog3.*`.

On the other hand, we do *not* use the `MID_DIR` macro in the indirect dependency statement for `prog3.obj`. The macro is not required here because *make* has already issued the command

```
cd  $(MID_DIR)
```

so that the current directory is `middle1`, where all the `prog3` files reside.

Returning to the line that reinvokes *make*, we see two other points of interest: we have specified a target, `prog3.obj`; and we have defined a new macro that is passed to the second invocation of *make*. When this second instance of *make* executes the command line

```
echo  $(COMM)
```

it expands the macro `COMM` to display the message

```
Compiling PROG3.C
```

before compiling the program. When the second instance of *make* has finished creating `prog3.obj`, it returns to the original process and the initial instance of *make* continues to create the target, `prog.exe`.

One question we might ask is: Why are all the dependencies for the `middle1` program placed in the `top` makefile? The reason is simple. It requires only a small amount of work to build `prog3.obj`, so we can use the existing makefile to perform all that is needed. A second reason is that we do not want to maintain duplicate copies of a makefile to produce the same results.

Now let's look at the same makefile for MAKE from Borland:

```
# Recursion using the same makefile (Borland)

# Define suffixes

.suffixes:
.suffixes:  .c  .obj  .exe

# Define macros

CFLAGS          =  -c  -w  -O2
MID_DIR         =  middle1
OBJS            =  prog1.obj   prog2.obj
```

```
EXEC_NAME       =  prog

# Define inference rule

.c.obj:
        $(CC)  $(CFLAGS)  $*.c

# Define target and dependencies

$(EXEC_NAME).exe:  $(OBJS) $(MID_DIR)\prog3.obj
        $(CC) -e$@ $(OBJS) $(MID_DIR)\prog3.obj

$(MID_DIR)\prog3.obj:  $(MID_DIR)\prog3.c
        cd  $(MID_DIR)
        $(MAKE) -f ..\makefile                          \
                COMM=PROG3.C prog3.obj
        cd ..

prog3.obj:      prog3.c
        echo  $(COMM)
        $(CC) $(CFLAGS) prog3.c
```

We've made a few changes to the makefile. First, we changed the two pseudotarget statements to lowercase (`.suffixes:`). Second, we also changed the warning level compiler option to lowercase (`-w`). The third change is the compiler option in the command line

```
$(CC) -e$@ $(OBJS) $(MID_DIR)\prog3.obj
```

that produces the target `prog.exe`.

You may notice, too, that we explicitly defined the inference rule. We did this because Borland's predefined inference rule uses `$&` (no path) and, in this case, we need path information.

The final change is the way we invoke Borland's MAKE with a macro defined on the command line:

```
$(MAKE) -f ..\makefile                  \
        COMM=PROG3.C prog3.obj
```

Borland's MAKE imposes a limitation on the definition of the macro: the string that follows the equal sign (=) cannot contain spaces, tabs, or quotation marks.

As we've just seen, the makefile for Microsoft's and IBM's NMAKE and the makefile for Borland's MAKE are basically the same. The changes we've made in the makefile for Borland's MAKE have to do with:

- a predefined pseudotarget (`.suffixes:` in place of `SUFFIXES:`)
- compiler options defined in the `CFLAGS` macro (`-w` in place of `-W2`, and `-O2` in place of `-O`)
- a different compiler option (`-e` as opposed to `-o`)
- a different method of passing a macro from the command line to *make*

Finally, let's look at the UNIX version of this makefile.

```
# Recursion using the same makefile (UNIX Systems)

# Define suffixes

.SUFFIXES:
.SUFFIXES:      .c  .o

# Define macros

MID_DIR         =  middle1
OBJS            =  prog1.o  prog2.o
EXEC_NAME       =  prog

# Define target and dependencies

$(EXEC_NAME):   $(OBJS) $(MID_DIR)/prog3.o
        $(CC) -o $@ $(OBJS) $(MID_DIR)/prog3.o

$(MID_DIR)/prog3.o:  $(MID_DIR)/prog3.c
        cd  $(MID_DIR) ;                              \
        $(MAKE)  -f  ../makefile                      \
                COMM="Compiling PROG3.C" prog3.o

prog3.o:        prog3.c
        echo  $(COMM)
        $(CC) $(CFLAGS) -c prog3.c
```

A quick comparison reveals that only a few changes were needed to convert the makefile from use with Microsoft's and IBM's NMAKE to use on UNIX systems. Here is what we did:

- removed the `.exe` extension from the target name (`EXEC_NAME`)
- treated `cd $(MID_DIR)` and the act of reinvoking *make* as one process (see Chapter 7)
- changed the object file extensions from `.obj` to `.o`
- changed the backslashes (\) in directory paths to forward slashes (/)

The rest of the makefile is untouched.

When we look at these three makefiles side by side, we see that they are essentially the same. We may need to make only a few, relatively simple changes to adapt a makefile that works for one version of *make* for use with another version, thus highlighting one of the valuable features of *make*.

Using Environment Variables in Makefiles

Makefiles can access and use environment variables. But just what is an environment variable? We know that a variable is a name to which a value is assigned and that the assigned value can be changed at will as needed. The environment is the collection of definitions required by the operating system to perform basic tasks, such as displaying a prompt and searching directories for command files.

Thus, an environment variable is a variable that is defined in and controls some aspect of the system's environment. This means that each program that runs on the system is aware of and can use the environment information that the variable provides.

To determine what your system's environment consists of, enter `set` at the command line. The `set` command displays all the environment variables known to your system. For example, a list of known variables we might see if we entered `set` on a DOS machine is

```
COMSPEC=c:\command.com
PATH=c:\;c:\dos
```

or on an OS/2 machine:

```
COMSPEC=c:\os2\cmd.exe
PATH=c:\os2;
```

In UNIX systems you can use `set` or `env` to check the system's environment. Chapter 7 and the reference manual for your operating system discuss the environment variables it uses.

To create a new variable in the environment, we simply enter the `set` command followed by the variable name and the value we want to assign to it in the form:

```
set VARIABLE_NAME=VALUE
```

Sometimes we want to change the value of an environment variable the system already knows about, such as `PATH`. Sometimes we want to create a new environment variable because we use a program that expects to find such an environment variable. For example, the environment does not know about `ROOTDIR`, but a program could look for `ROOTDIR` and take the appropriate actions when it finds it. Before we use such a program, we must create a new environment variable, `ROOTDIR`.

In makefiles, an environment variable can be used very much like a macro. To illustrate, for our DOS example let's create an environment variable called `ROOTDIR` that we can use in our makefile. We create the variable by entering at the command line:

```
set ROOTDIR=\root
```

If we look at the DOS environment list again (using `set`), we see the following:

```
COMSPEC=c:\command.com
PATH=c:\;c:\dos
ROOTDIR=\root
```

Now that we have set a new environment variable, let's look at a makefile that uses the variable:

```
# Using environment variables (Microsoft)

# Define suffixes

.SUFFIXES:
```

```
.SUFFIXES: .c .obj .exe

# Define macros

CFLAGS          =  -c  -W2  -O
MID_DIR         =  middle1
OBJS            =  prog1.obj  prog2.obj
EXEC_NAME       =  prog

# Define target and dependencies

$(ROOTDIR)\$(EXEC_NAME).exe:  $(ROOTDIR)\$(OBJS)    \
                              $(MID_DIR)\prog3.obj
        cd  $(ROOTDIR)
        $(CC) -o $@ $(OBJS) $(MID_DIR)\prog3.obj

$(MID_DIR)\prog3.obj:  $(MID_DIR)\prog3.c
        cd  $(MID_DIR)
        $(MAKE) -f $(ROOTDIR)\makefile              \
                COMM="Compiling PROG3.C"  prog3.obj
        cd  $(ROOTDIR)

# Define indirect dependencies

$(ROOTDIR)\prog1.obj:   prog1.c

$(ROOTDIR)\prog2.obj:   prog2.c

prog3.obj:              prog3.c
        echo  $(COMM)
        $(CC) $(CFLAGS) prog3.c
```

We've changed our earlier makefile in several places by using the environment variable `ROOTDIR`. Because *make* automatically includes all environment variables during execution, they can be referred to as if they were macros. Notice that the target in this makefile is stated as

```
$(ROOTDIR)\$(EXEC_NAME).exe
```

which uses the environment variable `ROOTDIR` as a macro in the same manner

as the user-defined macro EXEC_NAME.

The changes we've made to the makefile allow *make* to be used from anywhere in the system; we are no longer limited to issuing `nmake` (or `make`) from a single directory. By using an environment variable in our makefile, we've added again to the flexibility and power of *make*.

Let's look at this makefile modified for the MAKE command from Borland:

```
# Using environment variables (Borland)

# Define suffixes

.suffixes:
.suffixes:  .c  .obj  .exe

# Define macros

CFLAGS          =  -c  -w  -O2
MID_DIR         =  middle1
OBJS            =  prog1.obj  prog2.obj
EXEC_NAME       =  prog

# Define inference rules

.c.obj:
        $(CC)  $(CFLAGS)  $*.c

# Define target and dependencies

$(ROOTDIR)\$(EXEC_NAME).exe:  $(ROOTDIR)\$(OBJS)    \
                              $(MID_DIR)\prog3.obj
        cd  $(ROOTDIR)
        $(CC) -e$@ $(OBJS) $(MID_DIR)\prog3.obj

$(MID_DIR)\prog3.obj:  $(MID_DIR)\prog3.c
        cd  $(MID_DIR)
        $(MAKE)  -f $(ROOTDIR)\makefile            \
                 COMM=PROG3.C prog3.obj
        cd  $(ROOTDIR)
```

```
# Define indirect dependencies

$(ROOTDIR)\prog1.obj:   prog1.c

$(ROOTDIR)\prog2.obj:   prog2.c

prog3.obj:              prog3.c
        echo $(COMM)
        $(CC) $(CFLAGS) prog3.c
```

As with the earlier examples, only minor changes have been made to adapt the makefile for use with Borland's MAKE. These changes include the lowercase pseudotarget name (`.suffixes:`), the new values for the `CFLAGS` and `COMM` macros, and the compiler option in the command line (`-e$@`).

Here is the modified makefile for use on UNIX systems:

```
# Using environment variables (UNIX Systems)

# Define suffixes

.SUFFIXES:
.SUFFIXES: .c .o

# Define macros

CFLAGS          =  -c
MID_DIR         =  middle1
OBJS            =  prog1.o  prog2.o
EXEC_NAME       =  prog

# Define target and dependencies

$(ROOTDIR)/$(EXEC_NAME):        $(ROOTDIR)/$(OBJS)  \
                                $(MID_DIR)/prog3.o
        cd  $(ROOTDIR) ;                            \
        $(CC) -o $@ $(OBJS) $(MID_DIR)/prog3.o
```

```
$(MID_DIR)/prog3.o:              $(MID_DIR)/prog3.c
        cd  $(MID_DIR) ;                                    \
        $(MAKE)  -f $(ROOTDIR)/makefile                     \
                COMM="Compiling PROG3.C" prog3.o

# Define indirect dependencies

$(ROOTDIR)/prog1.o:     prog1.c

$(ROOTDIR)/prog2.o:     prog2.c

prog3.o:                prog3.c
        echo  $(COMM)
        $(CC) $(CFLAGS) prog3.c
```

We see several minor changes to the makefile: the `CFLAGS` options have been changed to comply with the options for the compiler; `.exe` has been removed; every occurrence of `.obj` has been changed to `.o`; we treated `cd $(MID_DIR)` and the act of reinvoking *make* as one process (see Chapter 7); and, except for line continuation indicators, all backslashes (\) have been changed to forward slashes (/). No other changes were made to the makefile.

One final note about environment variables. The PC versions of *make* provide the `e` option, which restricts the makefile from changing the value of an environment variable during execution. This means that if the makefile included the statement

```
ROOTDIR=c:\tmp
```

and we invoked NMAKE or MAKE with the `e` option, the line in the makefile would *not* be executed and the value of `ROOTDIR` would remain `\root` (that is, the value we assigned using the `set` command).

You can find out more about how your version of *make* handles environment variables by referring to the documentation for your operating system and compiler.

Macro Information Substitution

As we've seen in many of our example makefiles, it is common to define macros for files that share the same base name but have different extensions. In fact, we've frequently defined macros somewhat like

```
OBJ = prog.obj
EXEC_NAME = prog
```

Interestingly, when we've used the second macro in a makefile, we've added an extension to it, whereas we haven't needed to add anything to the first macro because its definition includes the extension. Thus, we might have a target/dependency line that looks like:

```
$(EXEC_NAME).exe:    $(OBJ)
```

What we have done, in effect, is to define two macros with nearly identical information. This begins to seem cumbersome as well as redundant. Fortunately, every version of *make* supports a feature that we call *macro information substitution*. Using this feature, we can eliminate one of the macros, yet provide the same functionality in the makefile. Here is an example showing this feature in action:

```
# Using Macro Information Substitution

# Define suffixes

.SUFFIXES:
.SUFFIXES: .c .obj .exe

# Define macros

CFLAGS    =  -c  -W2  -O
EXEC_NAME = prog.exe

# Define target information

$(EXEC_NAME): $(EXEC_NAME:.exe=.obj)
        $(CC) -o $@ $(EXEC_NAME:.exe=.obj)
```

In this example, we have defined `EXEC_NAME`, but not `OBJ`. Then, using macro information substitution in the target/dependency line and in the command line, we have informed *make* that we want to use the base information from the `EXEC_NAME` macro (`prog`), but want to replace the defined extension (`.exe`) with a different extension (`.obj`).

In this way, we can make full use of the power of macros without cluttering the makefile with redundant macro definitions. Keep this often-neglected feature in mind as you write your makefiles; you are sure to find many uses for macro information substitution in your development efforts.

Beware the Macro Happy Makefile

By now you might be thinking that macros are the greatest contribution to program development since disk storage replaced punched cards. We've certainly encouraged you to use macros: we've explained how they add flexibility to your makefile and we've shown numerous examples. However, with macros as with environment variables, too much of a good thing is... *not* a good thing.

So we want to take a moment here to caution you against writing "macro happy" makefiles. By this we mean makefiles in which nearly every item is defined as a macro, without thought as to the relevance or usefulness of the definition.

At some point, the overuse of macros will cause a makefile to be nearly unreadable. And, contrary to our underlying objective, a makefile in which macros are misused will be less, rather than more, adaptable for future enhancements.

Remember, the reason we define macros in a makefile is to provide a substitute name for an element that is subject to change. We've seen examples where we might want to use the same makefile with different compilers or with different compiler options or to build different targets from different dependencies. These are all appropriate uses for macros. In contrast, we derive no benefit from defining macros for fixed functional items, such as basic shell commands. Here is an example of what we mean by a "macro happy" makefile:

```
# Using environment variables (Microsoft)

# Define suffixes

.SUFFIXES:
.SUFFIXES: .c .obj .exe

# Define macros

CFLAGS          =  -c  -W2  -O
MID_DIR         =  middle1
OBJS            =  prog1.obj  prog2.obj
EXEC_NAME       =  prog
CD              =  cd
ECHO            =  echo

# Define target and dependencies

$(ROOTDIR)\$(EXEC_NAME).exe:  $(ROOTDIR)\$(OBJS)    \
                              $(MID_DIR)\prog3.obj
        $(CD)  $(ROOTDIR)
        $(CC) -o $@ $(OBJS) $(MID_DIR)\prog3.obj

$(MID_DIR)\prog3.obj:  $(MID_DIR)\prog3.c
        $(CD)  $(MID_DIR)
        $(MAKE) -f $(ROOTDIR)\makefile             \
                COMM="Compiling PROG3.C"  prog3.obj
        $(CD)  $(ROOTDIR)

# Define indirect dependencies

$(ROOTDIR)\prog1.obj:   prog1.c

$(ROOTDIR)\prog2.obj:   prog2.c

prog3.obj:              prog3.c
        $(ECHO) $(COMM)
        $(CC) $(CFLAGS) prog3.c
```

We can see that the first four macros defined at the top of the file provide us with the flexibility to apply this same makefile in a variety of circumstances. But the makefile and our development projects gain nothing from defining the `cd` and `echo` commands as macros.

Common Errors

Despite our best efforts, errors will inevitably creep in as we develop makefiles for the projects we manage. In this section, we try to anticipate some of the errors most commonly made. We show the messages these errors produce and explain what they mean. Ideally, by knowing about the most common errors in advance, we can avoid them.

Error:

`No arguments or description file.`

make cannot find a file called `makefile`. If the makefile has a name other than `makefile`, use the `f` option and the name of the file to inform *make* of the name of the makefile. For example, if you named the file `descript.mak` and you are using NMAKE from Microsoft or IBM, use the command line:

```
nmake  -f  descript.mak
```

If you are using Borland's MAKE, use:

```
make  -f  descript.mak
```

Error:

`Must be a separator on rules line ##.`

PC versions of *make* expect commands to be preceded by one or more tabs or spaces. You will see this message when the command starts in column 1 of the referenced line. In the UNIX systems we use, the command must be preceded by one or more tabs (*not* space characters).

Error:

`Don't know how to make` *xxxxxx*.

make does not have some piece of information it needs to make a target or a dependency. For example, the following makefile contains one target, `yyyy`, and one command:

```
yyyy:
        echo "This is target yyyy."
```

If you invoke *make* and ask it to build target `zzzz` with the command line

```
nmake  zzzz
```

make responds that it does not know how to make `zzzz`. This is because the target `zzzz` does not exist.

make produces this message in another context: when it cannot find a file for a dependency. For example, if the makefile contains the lines

```
prog.exe:   prog.c
        cl  prog.c
```

and `prog.c` is not in the current directory, *make* tells you it does not know how to make `prog.c`.

In the versions of *make* that support the pseudotarget `.SUFFIXES:`, be sure that your *make* knows of the suffixes you are using and that *make* has an inference rule to apply to your file. Otherwise, *make* will not know how to make `xxxxxx`. This requirement holds true, as well, for the variation of the pseudotarget that Borland's MAKE supports (`.suffixes:`).

Error:

xxxxxx `is up-to-date.`

make checked the date and time for the dependencies of `xxxxxx` (that is, the target being built) and found that none of the dependencies is newer than `xxxxxx`. If your list of dependencies is correct, this is a valid message. It means that *make* did not have anything to do. If you changed any of the files that the target `xxxxxx` depends on, then the file you modified does not appear in the dependency list for the target `xxxxxx`.

Error:

`syntax error: separator missing.`

make found a situation that does not follow the standard construct for a makefile:

```
macroname  =  macrovalue

target:  dependency1  dependency2 ...
        command1
            ⋮
        commandn
```

For example, in a makefile the lines

```
!ifndef  DEBUG
echo DEBUG is not defined.
!endif

yyyy:
        echo This is yyyy.
```

will produce this syntax error message. This is because the `echo` command within the `ifndef` directive does not follow a target. Remember that commands appear after targets like `yyyy:`.

Error:

`line ##: syntax error.`

Line *##* contains an error. If the referenced line in your makefile looks like a blank line, then the error is caused by tab characters; *make* will produce a syntax error for lines that contain only tab characters.

Error:

`syntax error: 'xxxxxxx' unexpected`

In the following example, *make* found a dependency line beginning with the target `prog.exe:`. The blank line that follows the dependency line

indicates the end of the command list; in this case, the command list was empty because of the intervening blank line.

```
prog.exe:   prog.c
        cl  prog.c
```

The command line that begins with `cl` is neither a target nor a part of the earlier command list. Therefore, *make* will not be able to interpret it, and the message will say that `'cl'` is unexpected.

Summary

We can extend the versatility and power of *make* by using recursion, environment variables as macros, and macro information substitution.

Recursion is a process in which *make* reexecutes itself. We can use *make* recursively in two ways:

- Use a different makefile for each instance of *make*.
- Use the same makefile and apply it to different targets for each instance of *make*.

Recursion allows us to define a new macro on any of the makefile command lines and to pass it to a recursive instance of *make*. Finally, using *make* recursively relieves us of the need to maintain multiple copies of a makefile to perform the same tasks in multiple directory locations or on multiple targets.

All versions of *make* support the predefined macro `MAKE`, thus facilitating recursive use of the utility.

make automatically includes all environment variables during execution. Thus, once an environment variable has been defined, it can be used in a makefile as if the variable were a predefined macro.

We use the `set` command to display all environment variables currently known to the system. We define a new variable on the command line by assigning a value to a name in the form:

```
set VARIABLE_NAME=VALUE
```

One useful application of this technique is to set a directory path so that *make* can be used from anywhere in the system.

Macro information substitution allows information within a macro to be changed (for evaluation purposes) during the processing of a section of a makefile. This feature eliminates the need to define multiple macros that provide the same base information.

Chapter 5

The MAKE Utility on DOS Systems

In previous chapters we looked at concepts and strategies that, for the most part, apply universally to all versions of *make*. Here we focus our attention on the *make* utility on DOS systems; specifically, NMAKE from Microsoft and MAKE from Borland.

We have structured this chapter on two premises. First, that you are interested in how *make* works on a DOS system and, second, that you have focused on either Microsoft's NMAKE or Borland's MAKE. Thus, the chapter is organized in two essentially parallel sections — one for each version of the *make* utility on DOS — and a concluding summary that covers both versions. If you read both the NMAKE section and the MAKE section, you will notice some duplication of material.

NMAKE on DOS

We begin this section by showing how to invoke NMAKE. Then we explore how some of the topics we met earlier — such as options, dependencies, continuation lines, and environment variables — apply specifically to Microsoft's NMAKE. New topics introduced in this section include preprocessor directives, input files and in-line response files, and the `tools.ini` file. We conclude with some tips for using NMAKE to develop applications for a graphical user interface (GUI) environment.

Invoking NMAKE

Here is the format for invoking NMAKE on the command line:

```
nmake [options] [macro_definitions] [targets]
```

The brackets ([ ]) around a field name indicate that the field is optional. Thus, all three fields are optional for NMAKE.

Command Line Options

NMAKE supports an extensive list of command line options. If you are familiar with other versions of *make*, some of these options may look familiar. However, a number of NMAKE options differ from those supported in other versions of *make*.

NMAKE is not case sensitive with respect to options; that is, NMAKE treats `a` and `A` as the same option. Remember that when we specify an option on the command line, we must precede the option letter with a dash (`-`) or a forward slash (`/`).

Here are the options that NMAKE supports:

`?`	Display short help
`a`	Build all targets
`b`	Execute a dependency even if time stamps are equal
`c`	Suppress messages
`d`	Display modification dates
`e`	Let environment variables override macro definitions
`f` *filename*	Specify makefile to be *filename*
`help`	Display help information
`i`	Ignore exit/error codes
`k`	Build unrelated parts of dependency tree if command fails
`m`	Swap NMAKE to disk to save memory space
`n`	Display commands but do not execute them

`nologo`	Suppress copyright message
`p`	Print values of defined macros and pseudotargets
`q`	Return exit code
`r`	Ignore `tools.ini` file
`s`	Suppress command display while executing
`t`	Change target modification dates
`v`	Cause macros to be inherited in NMAKE recursion
`x` *filename*	Specify destination for error messages
`z`	Internal to Programmer's Work Bench

Preprocessing Directives

A preprocessing directive is a command or an instruction that allows us to conditionally execute other commands or to conditionally select parts of a makefile. When NMAKE reads a makefile, it processes any preprocessing directives it finds before it executes commands associated with targets.

A preprocessing directive begins with an exclamation mark (`!`) and must appear at the beginning of the line. That is, we must place the `!` in the first column on the line. However, after the `!` we can use blank spaces and/or tabs, if we wish, to enhance readability of the file.

Here are the preprocessing directives that NMAKE supports:

`cmdswitches`	Change options in NMAKE
`error` *message*	Display error message
`include <`*filename*`>`	Insert another file as part of this makefile; NMAKE searches for the file in the directories specified in the environment variable `INCLUDE`
`include "`*filename*`"`	Insert another file as part of this makefile; NMAKE searches for the file in the current directory
`if defined` *name*	Execute statements following `if` when *name* has been defined; same as `ifdef` (defined later) except it can be used in complex expressions

`if exist` *path*	Return TRUE if *path* exists; can be used in complex expressions
`if` *expr*	Execute statements following `if` when *expr* evaluates to nonzero
`ifdef` *name*	Execute statements following `ifdef` when *name* has been defined as a macro
`ifndef` *name*	Execute statements following `ifndef` when *name* has not been defined
`else`	Execute statements following `else` when *expr* in the preceding `if`, or *name* in the preceding `ifdef` or `ifndef` evaluates to zero
`elseif` *expr*	Perform the same functionality as though an `else` directive were juxtaposed on the same line
`elseifdef` *name*	Perform the same functionality as though an `else` directive were juxtaposed on the same line
`elseifndef` *name*	Perform the same functionality as though an `else` directive were juxtaposed on the same line
`endif`	Mark the end of an `if`, `ifdef`, or `ifndef` directive
`message` *text*	Display the message text to the standard output and continue execution
`undef` *name*	Undefine *name*

We can use conditional operators with the `if`, `elseif`, and `else` preprocessing directives. The conditional operators that NMAKE supports are the same as those supported in the C programming language. They are:

- Unary operators
  - `-` arithmetic negation
  - `~` bitwise complement
  - `!` logical NOT

- Binary operators

+	addition	>>	right shift
-	subtraction	<<	left shift
*	multiplication	&	bitwise AND
/	division	|	bitwise OR
%	modulus	^	bitwise XOR
&&	logical AND	>=	greater than or equal
||	logical OR	<=	less than or equal
>	greater than	==	equality
<	less than	!=	inequality

We can use four of the binary operators listed above to compare string expressions as well as numeric expressions. The operators that we can use for string comparison are `>=`, `<=`, `==`, and `!=`.

To see how we can apply preprocessing directives, let's start with a makefile from Chapter 3. Here is the makefile:

```
CFLAGS =  -W2  -O
OBJS   =  prog1.obj  prog2.obj
TARGET =  prog.exe

$(TARGET):  $(OBJS)
        $(CC)  -o  $@  $(OBJS)
```

Now let's modify this makefile to use directives. Here is what we want to achieve:

- Include another makefile (called `sample.mak`) as part of our makefile. The macros we want to use will be defined in this file.
- If the `DEBUG` macro has been defined, then add the `Zi` and `Od` options to `CFLAGS` (`Zi` turns source debugging on; `Od` tells `cl` to disable optimization).

Here is the makefile in its new form:

```
# Define suffixes

.SUFFIXES:
.SUFFIXES: .c .obj .exe
```

```
# Include the file sample.mak

!include  "sample.mak"

# Test for defined macro

!ifdef DEBUG
CFLAGS  =  $(CFLAGS)  -Zi  -Od
!endif

# Target and dependencies

$(EXEC_NAME).exe:  $(OBJS)
        $(CC) -o  $@  $(OBJS)
```

Let's look more closely at the new features of this makefile, beginning with the **include** directive. We have used the **include** directive

```
!include  "sample.mak"
```

to tell NMAKE to read the file **sample.mak** from the current directory into the current makefile. When NMAKE begins to process the makefile, it appears as if the contents of **sample.mak** were always present. Effectively, they are: the directive brings them in. Here are the contents of **sample.mak**:

```
# Define macros

CFLAGS          =  -c  -W2
OBJS            =  prog1.obj  prog2.obj
EXEC_NAME       =  prog
```

By using the **include** directive, we can easily create part of a makefile — a sequence of statements — that is common to various makefiles we use. Then, whenever we need this particular sequence of statements, we tell NMAKE to include the file that contains them.

Let's look now at the **ifdef** directive. Remember that this directive asks NMAKE whether a particular macro has been defined. For our immediate interest, the directive

```
!ifdef DEBUG
```

asks NMAKE if a macro named DEBUG has been defined.

We can define the macro DEBUG in the makefile:

```
DEBUG  =  YES
```

Or we can define the macro on the command line when we invoke NMAKE:

```
nmake  DEBUG=YES
```

If DEBUG is defined, NMAKE executes the statements between the `ifdef` and the `endif` directives. In our case, NMAKE would change CFLAGS to be defined as:

```
CFLAGS  =  -c  -W2  -Zi  -Od
```

It is important to note that CFLAGS is left justified on the line; that is, the C of CFLAGS appears below the `!`. We must begin CFLAGS in the first column of the line because it is a macro name, not a command. Remember that only commands are indented on the line (with one or more spaces or tabs).

If DEBUG has not been defined, then NMAKE does not execute the statements between the `ifdef` and the `endif` directives; CFLAGS remains as originally defined.

Let's consider using the DEBUG macro in a different way. With the `if` and the `endif` directives, we can explicitly test whether DEBUG has been defined to be YES (or any other value we want to test for):

```
!if "$(DEBUG)" == "YES"
      ⋮
!endif
```

We can use an `else` directive following the `if` directive:

```
!if "$(DEBUG)" == "YES"
      ⋮
!else
      ⋮
!endif
```

This variation means that when DEBUG is defined to be YES, NMAKE executes the statements between the if and the else directives. When DEBUG is defined to be some value other than YES (or is not defined at all), NMAKE executes the statements between the else and the endif directives.

Let's add our debug flags to CFLAGS in the first case; let's keep CFLAGS simple in the second case:

```
!if "$(DEBUG)" == "YES"
CFLAGS = -c -W2 -Zi -Od
!else
CFLAGS = -c -W2
!endif
```

Notice in these examples of the if directive that we use the notation $( ) around DEBUG and we use double quotation marks on both sides of the expression. (Contrast this notation with the earlier example of the ifdef directive, where we used neither. We will return to this distinction in a minute.)

We must use $( ) around DEBUG because we want the if directive to evaluate the *value* of DEBUG. Without $( ), if would compare the literal string "DEBUG" with the string "YES". The test, of course, fails — and is not what we intended in any event.

We must use the quotation marks because we have asked NMAKE to evaluate a comparison, and the values to be compared must be of the same type. In our case, the value we want to test, YES, is a string; this means that the other value, $(DEBUG), must also be a string. We tell if to evaluate the comparison of two strings by surrounding each value with string delimiters (i.e., quotation marks). Without the quotation marks, if would expect to compare two integers.

Now let's consider the syntax of the ifdef directive. Recall that we used neither the notation $( ) nor quotation marks around DEBUG. This is because ifdef determines only whether the macro name has been defined; it does not check for the value of the macro, nor does it require the macro name to be a particular type.

We can write more than one statement following an if or an else directive. For example, we may need to change the DOS environment variable INCLUDE to add a directory to the list of directories that the C preprocessor searches.

In our case, we want to add the \DEBUG directory to the INCLUDE environment variable when the macro DEBUG is assigned the value YES. Here

are the lines we need in our makefile:

```
!if "$(DEBUG)" == "YES"
CFLAGS = -c -W2 -Zi -Od
!       if [set INCLUDE=%INCLUDE%;\DEBUG]
!       endif
!else
CFLAGS = -c -W2
!endif
```

Recall from Chapter 4 that we use the DOS `set` command to change the value of an environment variable. For NMAKE to perform this operation, the command must be within an `if` directive, where it always evaluates as nonzero. The brackets around the command are required, and the information within them

```
set INCLUDE=%INCLUDE%;\DEBUG
```

is passed directly to `command.com`.

We conclude this `if` directive right away with an `endif` because we do not have any other statements to execute after the `set` command.

From this brief discussion of preprocessing directives, we've gained some insight into their usefulness in makefiles. Different circumstances will suggest other uses. For more information about this versatile feature, refer to the online help that Microsoft provides with NMAKE.

Expressions in Preprocessing

Microsoft's NMAKE supports the use of C-like expressions in the preprocessing of a makefile. This feature allows us to create complex expressions that will determine which of several alternative actions the makefile will perform.

Let's explore how we can use complex expressions in a makefile. For this example, here is what we want to achieve:

- Place the executable file in its own directory, which we will call `execute`. The directory name will be defined as a macro in this makefile.

- Determine the version type of the program being created. If we are creating a debug version of the program, we want to copy the source code into the directory **execute**, which will allow debugging. If we are creating a non-debug (or production) version, then we want to delete all source code programs from that directory.

Here is the makefile that will accomplish these objectives:

```
# Define suffixes

.SUFFIXES:
.SUFFIXES: .c .obj .exe

# Include the file sample.mak

!include  "sample.mak"

# Define macros

DIRECTORY = ..\execute

# Test for defined macro

!ifdef DEBUG
CFLAGS          =  $(CFLAGS)  -Zi  -Od
!endif

# Check to see if directory exists; create if it does not

!if !exist ($(DIRECTORY)) && !defined (DEBUG)
!   if [md $(DIRECTORY)]
!   endif
!endif

# Check to see what version we are creating

!ifdef DEBUG
!   message Copying source files
!   if [copy *.c $(DIRECTORY)]
```

```
!   endif
!else
!   if [if exist $(DIRECTORY)\prog1.c del *.c]
!   endif
!   message Deleting source code files
!endif

# Target and dependencies

$(DIRECTORY)\$(EXEC_NAME).exe:  $(OBJS)
        $(CC) -o  $@  $(OBJS)
```

Now let's examine this makefile in some detail. We define the macro `DIRECTORY` to identify the output directory where we want to store the executable file (and the source code for debug versions). After defining the macros, we check for the existence of the output directory; to do this, we use the `if exist` directive to form a Boolean expression.

Because `if exist` *path* is a Boolean expression, it evaluates to true or false.

```
!if !exist ($(DIRECTORY)) && !defined (DEBUG)
!   if [md $(DIRECTORY)]
!   endif
!endif
```

In our case, we want to perform an action if the output directory does not exist and the macro `DEBUG` is not defined, so we use the negation expression (`!`), which is not to be confused with the `!` that identifies a preprocessor directive in the makefile. If the negated expression evaluates true (that is, if the directory does not exist and `DEBUG` has not been defined), we create it with the DOS shell command `md`.

Next we determine the kind of executable file we are creating. If it is a debug version, then we want to copy the current source code files to `$(DIRECTORY)`. On the other hand, if we are creating a production version of the program, then we do not want the source code in that directory and we want to remove all source files (`*.c`) that are there. Let's look a little more closely at this section of the makefile.

```
!ifdef DEBUG
!   message Copying source files
!   if [copy *.c $(DIRECTORY)]
!   endif
!else
!   if [if exist $(DIRECTORY)\prog1.c del *.c]
!   endif
!   message Deleting source code files
!endif
```

First, we use the `ifdef` directive to determine whether a macro named `DEBUG` has been defined (perhaps on the command line where NMAKE was invoked). If the macro has been defined, we use the `message` directive to report the action we are going to take, which is to copy the source files to the output directory. However, if the macro has not been defined, we follow up with a complex expression that determines whether any source files exist in the output directory. If so, we remove the files and use the `message` directive again to report that action.

As this example shows, by judiciously applying preprocessor directives and complex expressions, we can make decisions within a makefile that determine the course of action the makefile will take. These techniques expand the power of *make*, helping us to accomplish with relative ease tasks that would otherwise be much more troublesome.

Special Cases

We know from earlier chapters that when we construct and modify makefiles, we may need more than one line to specify a list of dependencies or a command with its various parameters. NMAKE accepts a backslash (\) before the end of a line as a signal that the current line has not ended. This means that the next line is the continuation of the line that ends with \.

This convention is helpful with long lines. This convention can also present a problem. For example, what will NMAKE do if it sees the lines

```
cd  \
copy  autoexec.bat  autoexec.old
```

in a makefile? The first command tells NMAKE to change to the root directory (in DOS, the \ character is the root directory for a device). Notice that this command ends with \, which is the last item on the

line. Because \ means the root directory in DOS and NMAKE uses \ as a line continuation indicator, we have a problem. Specifically, NMAKE understands the command to be

```
cd  copy  autoexec.bat  autoexec.old
```

which is scarcely what we had in mind!

Fortunately, we have a way to tell NMAKE when a \ is not meant to be a continuation indicator. We place a caret (^) in front of the backslash

```
cd  ^\
copy  autoexec.bat  autoexec.old
```

and NMAKE sees two command lines, as we intended.

The ^ turns off the special meaning — continuation indicator — for the backslash. This function of ^ extends to other characters in NMAKE. For example, the newline character signals the end of a line. If we want to use the newline as part of a string, we can turn off the meaning of newline by using a caret:

```
STRING=All^
is well.
```

We've defined the macro `STRING` to have two physical lines. Because ^ precedes the newline, `STRING` contains a newline after '`All`'. When we display the contents of `STRING`, we'll see '`All`' on one line followed by '`is well.`' on the next line. If we do not use the ^ in front of the newline, NMAKE will understand the value of `STRING` to be '`All`'. However, it will not recognize '`is well.`' as anything meaningful and consequently will display an error message.

Alternatively, we can insert a newline in a string by specifying the newline character as `\n`. So the two physical lines we defined previously for `STRING` become one line in the macro definition:

```
STRING=All\nis well.
```

In this case, of course, we do not need to use the ^ to turn off any special meanings. When we display the contents of `STRING` using this definition, we see two lines as we did before.

Creating Dependencies Dynamically

In Chapter 2 we mentioned the need for performing dependency checking. We are ready now to explore that concept further.

We well know by now that an important use of NMAKE is to maintain program modules within a development project. We also recall (from Chapter 2) that a source file may include other files.

We sometimes call these other files *header files* because they contain information that typically appears at the beginning, or head, of a program file. We also refer to these files as *include files*, for the obvious reason. In C language programs, the header (or include) files usually are recognizable by the `.h` extension to the file name; in assembler, the `.inc` extension is used. An include file may contain `#define` statements (C language) or `equate` statements (assembly language) that convey information to the file in which it is included.

Depending on the project, it may be more the rule than the exception that program modules use include files. To illustrate how this affects dependencies in a makefile, let's look again at the makefile we used in Chapter 3:

```
CFLAGS =  -W2  -O
OBJS   =  prog1.obj  prog2.obj
TARGET =  prog.exe

$(TARGET):  $(OBJS)
        $(CC)  -o  $@  $(OBJS)

prog1.obj:  prog1.c

prog2.obj:  prog2.c
```

(The lines with `prog1.c` and `prog2.c` ordinarily would not appear in the makefile because of `.SUFFIXES:` and the predefined inference rules; we are showing them in this case to illustrate the need for dependency checking.)

Based on this makefile, the only dependency for `prog1.obj` is `prog1.c`. But suppose that `prog1.c` uses the include file `prog1.h`. The relevant line in the makefile

```
prog1.obj:  prog1.c
```

gives no indication that `prog1.h` has anything to do with this dependency.

Now let's say that we have changed `prog1.h` — but not `prog1.c` — since the last time our makefile produced the `.obj` file for `prog1.c`. Because `prog1.c` includes `prog1.h`, we want NMAKE to recompile `prog1.c` when either it or `prog1.h` is modified. But NMAKE does not know this. In this situation, NMAKE does not know to check the date and time of `prog1.h`. It checks the date and time of `prog1.obj` and of `prog1.c` only, and does not recompile `prog1.c` until `prog1.c` is modified.

An obvious solution to the apparent dilemma presents itself. We need simply to change our makefile to show that `prog1.obj` depends on both files:

```
prog1.obj:  prog1.c  prog1.h
```

It is clear now that `prog1.h` is a direct dependency for `prog1.obj`.

It is vital, as this example shows, that we state all the dependencies for every file we need to work with in the makefile.

The importance of this requirement cannot be overstated; however, it is not always easy to remember every include file in every program module. But consider this: what if we devise some way to have NMAKE help us to satisfy the requirement?

The tool we have in mind is a program that creates a correct — and complete — list of dependencies for the makefile. Still working with the example from Chapter 3, our program will have to read the two identified indirect dependencies, `prog1.c` and `prog2.c`, and look for include files within them. The output of the program will be the original makefile plus a dependency list for each `.obj` file. Then any time we modify either of the `.c` file, we will rerun the program to create an updated dependency list.

We have, in fact, developed such a tool: the program `depends` in Appendix I builds a dependency list from the dependencies stated in the makefile. The program reads the current makefile to collect the object file names (`prog1.obj` and `prog2.obj`). `depends` then reads each source program (it looks for `.c` and `.asm` files for a given `.obj` target), looking for `include` statements, and updates the makefile to reflect the current dependency list.

We must make some changes to our makefile if we want to use `depends` to accomplish automatic dependency checking. Here is the modified makefile:

```
CFLAGS  =  -c  -W2  -O

prog.exe:  prog1.obj  prog2.obj
        $(CC)  -o  $@  prog1.obj  prog2.obj

dependen:
        copy  makefile  makefile.old
        depends  -I.  -I%%ROOTDIR%%  >makefile
        $(MAKE)
```

Let's investigate the changes we made. First, we removed the dependency lines

```
prog1.obj:  prog1.c
prog2.obj:  prog2.c
```

because `depends` will generate them. We also created a pseudotarget, `dependen`, with three command lines:

```
copy  makefile  makefile.old
depends  -I.  -I%%ROOTDIR%%  >makefile
$(MAKE)
```

The `copy` command makes a backup copy of `makefile`. We save a copy of the makefile in `makefile.old` just in case something does not work the way we expect it to work. (Another reason is that `depends` reads the input from a default file named `makefile.old`. See Appendix I for information about naming the input file.)

The `depends` command executes the `depends` program using a few options. (The options are explained in Appendix I.) We redirect the output from `depends` (the original makefile and the dependency list) to the current makefile; that is, we overwrite the existing makefile. Finally, because the predefined macro `MAKE` expands to the `nmake` command, NMAKE is invoked recursively.

One note about the options for `depends`: we use `%%` around the environment variable `ROOTDIR` because `depends` expects to find an environment variable surrounded by %s, as in `%ROOTDIR%`. To NMAKE, % is a special character, so we must use two %s to mean one %. Using `-I%%ROOTDIR%%` allows us to pass `-I%ROOTDIR%` to `depends`.

Now, to build the dependency list, we invoke `nmake` on the command line and specify the target:

```
nmake  dependen
```

Here is the makefile after `depends` modifies it:

```
CFLAGS  =  -c  -W2  -O

prog.exe:  prog1.obj  prog2.obj
        $(CC)  -o  $@  prog1.obj  prog2.obj

dependen:
        copy  makefile  makefile.old
        depends  -I.  -I%%ROOTDIR%%  >makefile
        $(MAKE)

prog1.obj: prog1.c      \
                f:\c600\include\stdio.h         \
                .\prog1.h

prog2.obj: prog2.c      \
                f:\c600\include\stdio.h
```

`depends` determined that `prog1.c` includes two files, `stdio.h` and `prog1.h`, and that `prog2.c` includes only one, `stdio.h`.

We can see that the job of building the dependency list for a makefile is much easier when we use a program like `depends`.

Let's review how this works. From the command line we execute

```
nmake  dependen
```

which causes NMAKE to begin executing the makefile at the pseudotarget `dependen`. NMAKE then makes a backup copy of the makefile, creates the dependency list, and writes it to `makefile`. When NMAKE reaches the third command line for the pseudotarget, it reinvokes itself and begins executing the same makefile from the top. This second instance of NMAKE finds `prog.exe` to be the first target in the makefile, so it begins to do all the things necessary to build `prog.exe`. Note that in this instance NMAKE does not execute the commands associated with the pseudotarget because `dependen` is not required for `prog.exe`. NMAKE does, however, find the dependency lines for `prog1.obj` and `prog2.obj` because `depends` inserted them on the first instance of NMAKE.

NMAKE does not find `dependen` the second time because that target is not part of `prog.exe` and we do not explicitly refer to that pseudotarget. In this case, the only way we reach `dependen:` is by using the target name on the command line.

Input Files and In-Line Response Files

Remember the very first makefile we looked at in Chapter 1? Here it is again:

```
prog.exe:  prog1.obj  prog2.obj
        cl  -o  prog.exe  prog1.obj  prog2.obj

prog1.obj:  prog1.c
        cl  -c  -W2  -O  prog1.c

prog2.obj:  prog2.c
        cl  -c  -W2  -O  prog2.c
```

Notice that we linked the `.obj` files and produced the target `prog.exe` using the command line:

```
cl  -o  prog.exe  prog1.obj  prog2.obj
```

An alternative to using `cl` to link is using the `link` command explicitly:

```
link  prog1.obj  prog2.obj, prog.exe ;
```

Notice the semicolon (`;`) at the end of the `link` command line. The purpose of the semicolon is to instruct the linker to use default values for the link options instead of prompting the user for them. If we want to use only some (or none) of the default values, we simply include on the command line the options we want before (or instead of) the semicolon.

The following command line shows a case in which we have decided not to use all the default options:

```
link  prog1.obj  prog2.obj,  prog.exe,  ,  slibce ;
```

(The `link` command accepts the names of the `.obj` files separated by either blanks or plus signs (+).)

This `link` command line includes an option that was not present in the earlier case: `slibce`, which tells the linker which library to bring in to link with our object files.

Notice that we retained the semicolon at the end of the command line. Without it, the linker might prompt for other options that we haven't specified. However, because the semicolon is present, the linker knows to use default values for the rest of the link options.

Our example `link` command line is pretty compact, but command lines can easily become quite lengthy. Command lines in makefiles can be particularly bothersome because *make* requires a backslash (\) at the end of a line to indicate that the next line is a continuation of the current line.

The solution to dealing with lengthy, unwieldy command lines in a makefile is simple. Using our example, we place the parameters to the `link` command in a file. Then, in the makefile, we tell the linker to use the file instead of depending on a list of parameters on the command line.

To get an idea of how this works, let's put our link parameters into a file called `makefile.lnk`. Here they are again:

```
prog1.obj  prog2.obj,  prog.exe,  ,  slibce ;
```

We've just created an input file to the linker. (An input file is also referred to as a *response file* or an *autoresponse file.*) To have the linker read the input file, we precede the name of the input file with the "at" character (`@`). (Note that this use of `@` is distinct from the predefined macro `$@`.) This character tells the linker to use the information stored in the file and not to prompt for input. Now the `link` command line within the makefile looks like this:

```
link  @makefile.lnk
```

It's easy to see, even with our compact example, how helpful an input file can be. Its usefulness is even more evident when a large number of files and linker options must be used.

Now that we understand the basic concept of the input file to a linker, let's consider how we can make it even more useful. For example, what if we want to use a macro in an input file? We can do it: the linker is aware of environment variables.

Recall that in Chapter 4 we created an environment variable called ROOTDIR and showed how we could use it as if it were a macro. Here's an example of what our input file would look like using the environment variable as a macro:

```
prog1.obj  prog2.obj,  $(ROOTDIR)\prog.exe,  ,  slibce ;
```

Closely related to input files are in-line response files. You can see the similarity in the following example:

```
link  @<<makefile.lnk
prog1.obj  prog2.obj,  $(ROOTDIR)\prog.exe,  ,  slibce ;
<<keep
```

In the earlier example, we told the linker (by means of the @ character) to get its parameters from a file named makefile.lnk. In the example just above, we've inserted the notation << between the @ character and the input file makefile.lnk. This format indicates that the file makefile.lnk will be created from the items that follow, beginning on the next line and continuing up to the next occurrence of <<. A file created in this manner is called an *in-line response file.* If a file of the same name already exists (makefile.lnk in our example), it is overwritten by the input to the in-line response.

We can construct an in-line response with as many lines as we like. To do this, we simply add lines — without backslashes — and use the second occurrence of << to indicate that we have completed the in-line response:

```
link  @<<makefile.lnk
prog1.obj  prog2.obj
$(ROOTDIR)\prog.exe
nul
slibce ;
<<keep
```

Note that a file created by in-line response is a temporary file. After executing the makefile, NMAKE removes any temporary files that were created in the process. Now, we might want to save makefile.lnk for some future use. If we do, we must follow the << at the end of the in-line

response with the command `keep`. If we are not interested in saving the file to use again, we omit `keep`. (The default is `<<nokeep`, which is equivalent to `<<`.)

Using Command Level Processing in Makefiles

In an ideal world, all programs would execute to completion successfully every time. Alas, this is not an ideal world. Despite our best efforts, the programs we develop do fail from time to time, for one reason or another. Thus, we use exit (or return) codes to determine whether a program failed and, often, what caused the failure.

In the DOS environment, the command processor supports the `errorlevel` statement. With this mechanism, we can test a program's exit code and take appropriate action based on the result. Consider the following simple example:

```
runxyz
if errorlevel 1 doprtscn
```

Here we execute the program `runxyz`, then test its exit code with the `errorlevel` statement. If `runxyz` returns an exit code of 1 or higher, we execute a program that writes the contents of the screen to the printer. If the exit code is any other value (or if `runxyz` does not return an exit code), no other action is performed.

This technique can be used effectively in batch, or command, processing. Unfortunately, it does not work for makefiles. NMAKE does not provide any mechanism that allows for command level processing, such as testing a program's exit code. At least, not directly. But there is a way around this difficulty.

Let's say that we write a makefile in which we execute the program `runxyz`. The program fails. Now what?

Well, if we invoked NMAKE without the `i` option, NMAKE would stop when it received the exit code from `runxyz`, indicating the program had failed. On the other hand, if we used the `i` option (or the `.IGNORE:` pseudotarget), NMAKE would ignore the error reported by `runxyz` and continue processing the rest of the makefile.

Neither action suits our purpose. How then can we execute a program from within a makefile, determine whether the program worked correctly,

and have NMAKE take some alternative action based on the success or failure of the program?

The answer is to create a pseudotarget that uses the in-line response function we discussed in the previous section. With in-line response we can create a batch file that will perform the tasks we require: it will execute the program `runxyz`, test the exit code, and proceed as determined by the result of the test. Here is a segment of a makefile that illustrates this technique:

```
# Define pseudotargets, macros, etc.

      ⋮

# Define temp.bat as a dependency target

$(EXEC_NAME).exe: $(OBJS) temp.bat

      ⋮

# Execute the pseudotarget batch file (defined below)

   temp

      ⋮

# Define pseudotarget temp.bat --
# a batch file to perform command level processing

temp.bat:
   @echo Creating batch file  @<<$@
@echo off
runxyz
if errorlevel 1 doprtscn
<<keep
```

Don't be confused by the various `@` characters you see in this example. As it happens, we use the same character to perform three separate functions. In `@echo` and `@echo off`, the character performs its DOS shell function of

suppressing the display of the function that is being called; in this case, @ prevents each of those lines from being displayed. The second use of @ is to signal the presence of an input file (in this case, the input file is an in-line response). We described this function of @ on page 86. The third use of this character is in the predefined macro $@.

Now let's carry this example a step further. Suppose that runxyz can return either of two different error level codes, and we want to perform different operations depending on which error level the program returns. Before implementing this more complex solution, we must understand that error level values are cumulative in command processing. This means, for example, that if error level 2 is set, so is error level 1. With this in mind, here is the segment of a makefile that allows us to direct NMAKE to take alternative actions based on the return code from runxyz:

```
temp.bat:
   @echo Creating batch file @<<$@
@echo off
runxyz
if errorlevel 2 writeerr
if errorlevel 1 doprtscn
<<keep
```

In this example, if runxyz returns error level 2 or higher, both writeerr and doprtscn are executed, in that order. (Note that we tested for error level 2 before error level 1.) If runxyz returns error level 1 only (that is, not error level 2), then only doprtscn is executed. If neither error level is set, no other action is performed.

By combining the in-line response function of NMAKE with the concept of batch files, we can achieve the equivalent of DOS command level processing indirectly through our makefiles.

NMAKE and the tools.ini File

The latest version of the Microsoft C compiler provides for a special file, called tools.ini, which NMAKE can use. When we invoke NMAKE, before it begins to execute the makefile it first reads tools.ini (when the file is present) and uses the macros and inference rules it finds there. There is an exception to this general rule: if a makefile and tools.ini both contain

a definition for a macro with the same name or an inference rule with the same extensions, the definition in the makefile overrides the definition in `tools.ini`.

NMAKE looks for `tools.ini` in the current directory. If `tools.ini` is not in the current directory, we can use the `set` command to create an environment variable called `INIT`. (We learned about environment variables and the `set` command in Chapter 4.) We can define `INIT` to specify the path to the directory that contains initialization files, including `tools.ini`. Now NMAKE can find `tools.ini` by searching the directory that `INIT` specifies.

We can use `tools.ini` to customize default options for NMAKE. Suppose, for example, that we want to use a file with the extension `.inc` to create another file with the extension `.tex`. We can create an inference rule so NMAKE knows how to perform this task for us:

```
.inc.tex:
        include  <$*.inc  >$*.tex
```

If we need this rule in several different makefiles, we can omit the rule from the individual makefiles and add it to `tools.ini`:

```
[NMAKE]
.inc.tex:
        include  <$*.inc  >$*.tex
```

Thus, we can use `tools.ini` the same way we use the `include` directive, which we learned about earlier. Is one technique preferred over the other? In terms of results, no; they are the same. However, consider this: If we use the `include` preprocessor directive, we must not only create another file, we must also remember to put the `include` statement in the makefile. NMAKE, on the other hand, looks for and reads `tools.ini` automatically, requiring no extra effort on our part (unless we use the `r` option).

The online help information from Microsoft provides more specific details about the `tools.ini` file and how to use it.

Macro Precedence

A macro defined on the command line has the highest precedence in NMAKE. At the next level of precedence are macros defined in the makefile. A macro defined in the makefile has precedence over inherited macros. (Inherited macros are macros that are equivalent to every environment

variable. NMAKE makes a copy of environment variables and gives them uppercase names.) At the next level below inherited macros are macros defined in the `tools.ini` file, followed by predefined macros.

For example, if a macro name appears in `tools.ini`, then NMAKE will use the value for that macro from `tools.ini` as long as a macro with the same name does not appear either on the command line or in the makefile and is not an inherited macro.

Let's consider a specific case. `CFLAGS` is a predefined macro that we discussed in Chapter 3. NMAKE will use its predefined value only if `CFLAGS` does not appear on the command line, is not defined in the makefile, is not an inherited macro, and does not appear in `tools.ini`. This is because predefined macros have the lowest precedence in NMAKE.

Using NMAKE to Develop GUI Applications

More and more applications are being developed for graphical user interfaces (GUIs) such as Windows from Microsoft, which runs on DOS systems. As we demonstrate in this section, programmers who use Windows libraries and resources to develop software can combine those tools with NMAKE to create GUI applications.

In comparison to the basic DOS program development we've discussed so far, Windows-type development requires a different approach to creating the various objects and the final executable file. The Microsoft C/C++ Compiler and the Software Development Kit (SDK) provide many of the libraries that we need to create a Windows application.

If we wish to develop a Windows application, we must have:

- Windows compiled program(s)
- Windows libraries – `xLIBCW.lib`, where `x` is the memory model being used: `s` (small), `l` (large), and so on.
- Any resource files (`.res` files) that are a part of the executable
- Module definition file

You can find detailed information about these items in the Microsoft C/C++ Version 7.0 reference manuals. Here we want to highlight just a few important points:

- You must use the Microsoft C/C++ libraries for Windows when you link your application files.

- You must use the Microsoft Resource Compiler (RC) to define and compile any resource files that your application will use. The resulting resource file is appended to the end of the executable and is considered read-only by the program. (See Figure 5.1.) The resulting resource file can be created from multiple source files that represent different sections of the application.

- The linker uses the module definition file to determine what type of executable to create. The module definition file is the *only* way you can tell the linker that you want to create a Windows application. Without a module definition file, the linker creates only non-Windows applications.

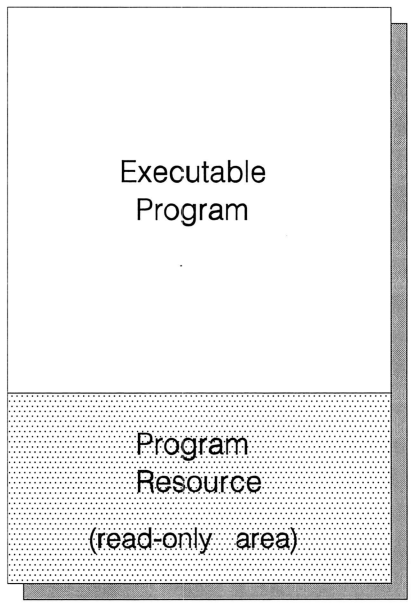


Figure 5.1. Windows GUI Format

Now that we have some understanding of the basic requirements, we are ready to look at an example. The following makefile (`winsamp.mak`) creates a window with text and two pull-down menus. One accesses the dialog

feature, and the other exits the program. (This example demonstrates many of the *make* and NMAKE functions we've already discussed, as well as the development of a GUI application.)

```
#
# Microsoft Makefile - Windows Example
#
# Mastering Make: A Guide to Building Programs on
# DOS, OS/2, and UNIX* Systems
# Tondo, Nathanson, Yount
#
# Prentice Hall (C) 1994
#

#
# Define suffixes
#

.SUFFIXES:
.SUFFIXES: .c .obj .exe .rc .res .dlg .def

#
# Define macros
#

CC      = cl
CINC    = -I.
LINK    = link
LIBS    = slibcw
RC      = rc
RCFLAGS = -r
TARGET  = winsamp.exe

#
# Test for defined macro
#

!ifndef DEBUG
DEBUG=yes
```

```
!endif

#
# Define macros for compiling and linking
#

!if "$(DEBUG)" == "yes" || "$(DEBUG)" == "YES"
CFLAGS  = -c -W3 -G2sw -Zp -W3 -AS -Zid -Od
LFLAGS  = /co
!else
CFLAGS  = -c -W3 -G2sw -Zp -W3 -AS -O
LFLAGS  =
!endif

#
# Define inference rules
#

.c.obj:
        $(CC) $(CFLAGS) $(CINC) $*.c

.rc.res:
        $(RC) $(RCFLAGS) $*.rc

#
# Define objects used to create the target
#

OBJS    = $(TARGET:.exe=.obj)

#
# Define the target to build
#

$(TARGET): $(OBJS) $(TARGET:.exe=.def)  \
                   $(TARGET:.exe=.res)
        $(LINK) $(LFLAGS) @<<$*.lnk
$(OBJS)
$@
```

```
NUL
$(LIBS)
$(TARGET:.exe=.def)
<<nokeep
        $(RC) -t $(TARGET:.exe=.res) $@

#
# Define module definition file used to create executable
#

$(TARGET:.exe=.def):
        @echo $@ @<<$*.def
NAME         WinSamp

DESCRIPTION  'Mastering Make Example - Microsoft'

EXETYPE      WINDOWS

CODE         PRELOAD MOVEABLE DISCARDABLE
DATA         PRELOAD MOVEABLE MULTIPLE

HEAPSIZE     1024
STACKSIZE    5120
<<keep

#
# Define indirect dependencies
#

$(TARGET:.exe=.obj): $(TARGET:.exe=.c)

$(TARGET:.exe=.res): $(TARGET:.exe=.rc)  \
                     $(TARGET:.exe=.dlg)
```

This is certainly the most complex makefile we've seen so far. Let's take a closer look at its various parts.

We begin by clearing NMAKE's built-in list of suffixes, then use the `.SUFFIXES:` pseudotarget to tell NMAKE what suffixes we want to use. In

the next three sections we define the various macros we plan to use in our makefile, such as the library to be linked in and the name of the target file. Because we are interested in whether the makefile will build a debug or a production version of the program, we test to see if the macro `DEBUG` is defined. If `DEBUG` is not defined, then the definition of the macro becomes `yes`. We will use this macro to determine whether we are going to build a debug or production version of the program. The result of the test allows us to set the correct values for the other compile- and link-related macros.

Next, we want to make sure that NMAKE knows how to create object (`.obj`) and resource (`.res`) files, so we define inference rules for them using some of the suffixes we defined at the beginning of the makefile. Finally, after defining the macro for the object files, we reach the target that we want to build. Notice that we include as dependencies not only the object file `winsamp.obj` (macro `OBJS`), but also the resource file `winsamp.res` (macro `$(TARGET:.exe=.res)`) and the module definition file `winsamp.def` (macro `$(TARGET:.exe=.def)`).

Now we come to the heart of this makefile. You may wonder why we have included the module definition file (`.def`) in the dependency list for an executable. Before we address that question, notice first that after defining the target, we use NMAKE's in-line response feature to perform two separate and distinct functions. In the first instance, we use in-line response to provide information to the linker. (We saw this usage earlier in this chapter.) But the second instance — ah, this is what catches our attention. Here we use in-line response to define a pseudotarget that effectively causes NMAKE to create the module definition file for us! This strategy assures us that two conditions — critical to our Windows application — are met:

- The module definition file is present.
- The module definition file is current.

Somewhat as an anticlimax, we conclude the makefile by defining the indirect dependencies for the object file `winsamp.obj` and the resource file `winsamp.res`.

Notice that the `$(TARGET:.exe=.dlg)` file is a part of the direct dependency for `$(TARGET:.exe=.res)`. This is because when the resource file is compiled, it includes all dialog files (`.dlg`) as part of the source file. Refer to the Windows sample source file (`.rc`) to see how the include process occurs. We must define the dialog file as we would any include file for a source file.

As we see from this example, no mystery attaches to using NMAKE to create a GUI application. The makefiles we write to build Windows applications are very much like any other makefile. We use the same fundamentals and similar techniques. As in any development project, we need only take care that our program conform to the requirements of the environment.

The Windows example we have just discussed presents a single point of view of how best to use NMAKE in a Windows development effort. Refer to Microsoft's user documentation for more information about Windows development and C applications. The complete source code for this example appears in Appendix F.

Borland's MAKE on DOS

We begin this section by showing how to invoke MAKE. Then we explore how some of the topics we met earlier — such as options, dependencies, continuation lines, and environment variables — apply specifically to Borland's MAKE. New topics introduced in this section include preprocessor directives, input files and in-line response files, and the `builtins.mak` file. We conclude with some tips for using Borland's MAKE to develop applications for a graphical user interface (GUI) environment.

Borland's C++ Version 3.1 provides an interactive development environment (IDE). The IDE automatically generates a makefile for the current project; however, this makefile can be used only within the IDE. If we want to work with this makefile outside the IDE, we can; but first we must run a program called `prj2mak`. The online documentation that Borland provides explains how to use this program.

Invoking MAKE

Here is the format for invoking MAKE on the command line:

```
make  [options]  [targets]
```

The brackets (`[ ]`) around a field name indicate that the field is optional. Thus, the two fields are optional for MAKE.

Command Line Options

MAKE supports an extensive list of command line options. If you are familiar with other versions of *make*, some of these options may look familiar. However, a number of MAKE options differ from those supported in other versions of *make*.

MAKE is case sensitive with respect to options; that is, MAKE treats `s` and `S` as two different options. Remember that when we specify an option on the command line we must precede the option letter with a dash (`-`) or a forward slash (`/`). Notice that when an option takes an argument we do not put any space between the option and the argument. (One exception is `f`; see below.)

Here are the options that MAKE supports:

`?`	Display help information
`a`	Perform automatic dependency check on all object files
`B`	Build all targets regardless of time stamp (file dates)
`d`*directory*	Identify where to write the swap file; *directory* can include a drive letter
`D`*identifier*	Define the value of *identifier* to be 1
`D`*identifier*`=`*string*	Define the value of *identifier* to be *string*
`e`	Let environment variables override macro definitions
`f`*filename*	Specify the makefile to be *filename*; space between `f` and the filename is optional (`f` *filename*)
`h`	Display help information
`i`	Ignore exit/error codes
`I`*directory*	Search for include files in *directory*
`K`	Retain any temporary file(s) created by MAKE

`m`	Display time and date stamp for each file as the file is being processed
`n`	Display commands but do not execute them
`N`	Change the behavior of in-line response processing to be compatible with NMAKE from Microsoft
`p`	Display all macros and inference rules before processing the makefile
`r`	Ignore rules that are defined in the `builtins.mak` file
`s`	Suppress command display while executing
`S`	Swap MAKE out of memory while executing commands
`U`*identifier*	Undefine *identifier* if previously defined
`W`	Write the current specified nonstring options to `make.exe`

Preprocessing Directives

A preprocessing directive is a command or an instruction that allows us to conditionally execute other commands or to conditionally select parts of a makefile. When MAKE reads a makefile, it processes any preprocessing directives it finds before it executes commands associated with targets.

A preprocessing directive begins with an exclamation mark (`!`) and must appear at the beginning of the line. That is, we must place the `!` in the first column on the line. However, after the `!` we can use blank spaces and/or tabs, if we wish, to enhance readability of the file.

Here are the preprocessing directives that MAKE supports:

`error` *message*	Display error message
`include <`*filename*`>`	Insert another file as part of this makefile; MAKE searches for the file in the directories specified in the environment variable `INCLUDE`
`include "`*filename*`"`	Insert another file as part of this makefile

`if` *expr*	Execute statements following `if` when *expr* evaluates to nonzero
`elif` *expr*	Execute statements following `elif` when the preceding `if` evaluates to zero and this *expr* evaluates to nonzero
`else`	Execute statements following `else` when *expr* in the preceding `if` (and in the preceding `elif`, when present) evaluates to zero
`endif`	Mark the end of an `if` directive
`undef` *name*	Undefine *name* (same as the `U`*identifier* option on the command line)

We can use conditional operators with the `if`, `elif`, and `else` preprocessing directives. The conditional operators that MAKE supports are the same as those supported in the C programming language. They are:

- Unary operators
  - `-` arithmetic negation
  - `~` bitwise complement
  - `!` logical NOT
- Binary operators

`+`	addition	`>>`	right shift
`-`	subtraction	`<<`	left shift
`*`	multiplication	`&`	bitwise AND
`/`	division	`|`	bitwise OR
`%`	modulus	`^`	bitwise XOR
`&&`	logical AND	`>=`	greater than or equal
`||`	logical OR	`<=`	less than or equal
`>`	greater than	`==`	equality
`<`	less than	`!=`	inequality

- Ternary operator
  - `? :` conditional expression

We can use four of the binary operators listed above to compare string expressions as well as numeric expressions. The operators that we can use for string comparison are `>=`, `<=`, `==`, and `!=`.

To see how we can apply preprocessing directives, let's start with a variation of a makefile from Chapter 3. We've adapted the makefile to work with Borland's MAKE. Here is the modified makefile:

```
CFLAGS          =  -c -ms
OBJS            =  prog1.obj  prog2.obj
TARGET          =  prog.exe

$(TARGET):      $(OBJS)
        $(CC)  -e$@  $(OBJS)
```

Now let's modify this makefile to use directives. Here is what we want to achieve:

- Include another makefile (called `sample.mak`) as part of our makefile. The macros we want to use will be defined in this file.
- Use the `$d` predefined macro (which tests for a defined macro) to determine if some action should be performed (see next item).
- If the `DEBUG` macro has been defined, add the `-O-` and `-v` compiler options to `CFLAGS` (`-O-` indicates that we want to disable optimization; `-v` turns source debugging on.)

Here is the makefile in its new form:

```
# Define suffixes

.suffixes:
.suffixes:  .c  .obj  .exe

# Include the file sample.mak

!include  "sample.mak"

# Test for defined macro

!if $d(DEBUG)
CFLAGS          =  $(CFLAGS)  -O-  -v
!endif
```

```
# Target and dependencies

$(EXEC_NAME).exe:  $(OBJS)
        $(CC)  -e$@  $(OBJS)
```

Let's look more closely at the new features of this makefile, beginning with the `include` directive. We have used the `include` directive

```
!include  "sample.mak"
```

to tell MAKE to read the file `sample.mak` into the current makefile. When MAKE begins to process the makefile, it appears as if the contents of `sample.mak` were always present. Effectively, they are: the directive brings them in. Here are the contents of `sample.mak`:

```
# Define Macros

CFLAGS          =  -c  -ms
OBJS            =  prog1.obj  prog2.obj
EXEC_NAME       =  prog
```

By using the `include` directive, we can easily create part of a makefile — a sequence of statements — that is common to various makefiles we use. Then, whenever we need this particular sequence of statements, we tell MAKE to include the file that contains them.

Let's look now at how we've used the `if` directive. Remember that this directive tells MAKE to evaluate an expression; that is, to determine whether the expression is true (the return value is nonzero) or false (the return value is zero). For our immediate interest, the directive statement

```
!if $d(DEBUG)
```

asks MAKE whether a macro named `DEBUG` has been defined. (Notice that the predefined macro `$d` uses a `d` — not a `D` — and should not be confused with the `D`*identifier* option from the command line. Remember that MAKE is case sensitive.)

We can define the macro `DEBUG` in the makefile:

```
DEBUG  =  3
```

Or we can define the macro on the command line using the `D`*identifier* option:

```
make  -DDEBUG=3
```

If DEBUG is defined, MAKE executes the statements between the if and the endif directives. In our case, MAKE would change CFLAGS to be defined as:

```
CFLAGS  =  -c -ms -O- -r-
```

It is important to notice that CFLAGS is left justified on the line; that is, the C of CFLAGS appears below the !. We must begin CFLAGS in the first column of the line because it is a macro name, not a command. Remember that only commands are indented on the line (with one or more spaces or tabs).

If DEBUG is not defined, then MAKE does not execute the statements between the if and the endif directives; CFLAGS remains as originally defined.

Let's consider using the DEBUG macro in a different way. With the if and the endif directives, we can explicitly test whether DEBUG has been defined to be 3 (or any other value we want to test for):

```
!if $(DEBUG) == 3

      ⋮

!endif
```

We can use an else directive following the if directive:

```
!if $(DEBUG) == 3

      ⋮

!else

      ⋮

!endif
```

This variation means that when DEBUG is defined to be 3, MAKE executes the statements between the if and the else directives. When DEBUG is defined to be some value other than 3 (or has not been defined at all), MAKE executes the statements between the else and the endif directives.

Let's add our debug flags to CFLAGS in the first case; let's keep CFLAGS simple in the second case:

```
!if $(DEBUG) == 3
CFLAGS  =  -c  -ms  -O-  -r-
!else
CFLAGS  =  -c  -ms
!endif
```

Notice in these examples of the `if` directive that we use the notation `$( )` around `DEBUG`. This is because we are interested in the value of the macro `DEBUG`, and the notation that tells MAKE to use the value of a macro is `$( )`.

Now let's consider a variation on our earlier directive statement:

```
!if !$d(DEBUG)
```

With this `if` directive using the `$d` predefined macro, we are asking MAKE to evaluate whether the `DEBUG` macro is defined. More specifically, by using the syntax shown, we are checking for the nonexistence of the macro (the second `!` in the statement is a NOT operator). In other words, if `DEBUG` is not defined, MAKE will return a true (nonzero) value to the preprocessing directive.

As with any `if` statement, we can use the results of this evaluation to conditionally perform some specific task. In the following example, the action MAKE takes is based on whether the macro `DEBUG` has or has not been defined:

```
!if !$d(DEBUG)
DEBUG  =  3
!endif
```

If the macro already is defined, the expression evaluates to false, and MAKE ignores the new definition given between the `if` and `endif` directives. However, if the macro is not defined, the condition given to the `if` directive evaluates to true, and MAKE defines the macro with the value `3`.

We can write more than one statement following an `if` or an `else` directive. For example,

```
!if $d(DEBUG)
CFLAGS  =  -c  -O-  -r-  -v
MESSAGE = "DEBUG has been defined"
!       if !$d(MODEL)
CFLAGS  =  -ms  $(CFLAGS)
```

```
!       else
CFLAGS  =  -m$(MODEL)  $(CFLAGS)
!       endif
!else
CFLAGS  =  -c
MESSAGE = "DEBUG has NOT been defined"
!       if !$d(MODEL)
CFLAGS  =  -ms  $(CFLAGS)
!       else
CFLAGS  =  -m$(MODEL)  $(CFLAGS)
!       endif
!endif
```

In this example we are testing first whether the `DEBUG` macro has been defined. If `DEBUG` has been defined, we assign one set of compiler options to `CFLAGS` and we assign a string to `MESSAGE`; otherwise, we define `CFLAGS` for a different set of options, and we assign a different string to `MESSAGE`. We test whether the `MODEL` macro has been defined in both cases.

Notice we have an `if` within an `if` (nested `if` statements) when we test whether the `MODEL` macro has been defined. In this case, if `MODEL` has not been defined, we add the `ms` compiler option to the value previously assigned to the `CFLAGS` macro; otherwise, the compiler option we add reflects the value already assigned to `MODEL`.

This example highlights two additional points of interest. First, each `if` directive must conclude with its own corresponding `endif` directive. Second, as long as we place the `!` in the first column of the line, we can use spaces or tabs to enhance readability.

Now let's make some more changes to the makefile to further illustrate the use of directives. This time our criteria are:

- Look in the current directory for the source files.
- Put the object files in a subdirectory called `sample2` (we will use the `o` compiler option to accomplish this).
- If the `DEBUG` macro is defined, change the compiler options to include debug information. Otherwise, use the value of `CFLAGS` as defined in the include file `sample.mak`.

To meet these criteria, we change our makefile so that it looks like this:

```
# Define suffixes

.suffixes:
.suffixes:  .c  .obj  .exe

# Define macros

.path.c   =  .
.path.obj =  .\sample2

OBJS = $(.path.obj)\prog1.obj $(.path.obj)\prog2.obj

# Include the file sample.mak

!include     "sample.mak"

# Test for defined macro

!if $d(DEBUG)
CFLAGS    =  $(CFLAGS)  -O-  -r-  -v
!endif

# Define inference rules

.c.obj:
        $(CC) $(CFLAGS) -o$(.path.obj)\$*.obj $*.c

# Target and dependencies

$(EXEC_NAME).exe:  $(OBJS)
        $(CC)  -e$@  $(OBJS)
```

Now let's look at the changes we've made. First, we added two `.path` statements. These statements tell MAKE the paths to use when searching for files with the `.c` extension and the `.obj` extension; specifically, the `.c` files for this makefile are in the current directory, and the `.obj` files are in a subdirectory called `sample2`.

Next, we changed the `OBJS` macro. We redefined `OBJS` to tell MAKE that the path for the object files is not the current directory, but is the

directory defined for `.path.obj`.

Finally, we changed the compiler command line associated with the inference rule for building `.obj` files from `.c` files. Now when MAKE compiles a `.c` file, it places the output (that is, the `.obj` file) in the directory defined for `.path.obj`.

From this brief discussion of preprocessing directives, we've gained some insight into their usefulness in makefiles. Different circumstances will suggest other uses. For more information about this versatile feature, refer to the documentation that Borland provides for MAKE.

Special Cases

Unlike other versions of the *make* utility, Borland's MAKE does not support escape sequences. This fact sets up an interesting situation.

We already know that a backslash (\) at the end of a line tells MAKE that the next line in the makefile is a continuation of the line on which the \ occurs. Suppose, however, that we want to copy a file created in our makefile to the root directory of the system, which also is represented by the character \. How does MAKE know the difference between the case where \ indicates a continuation line

```
prog1.obj: prog1.c     \
        prog1.h
```

and the case where it does not:

```
copy  $<  \
```

The answer, fortunately, is simple: we can use a double backslash (\\) to tell MAKE that it should see a literal \ and not a line continuation indicator. Thus, when MAKE sees

```
copy  $<  \\
```

it knows to copy the file represented by `$<` to the root directory.

Creating Dependencies Dynamically

In Chapter 2 we mentioned the need for performing dependency checking. We are ready now to explore that concept further.

We well know by now that an important use of MAKE is to maintain program modules within a development project. We also recall (from Chapter 2) that a source file may include other files.

We sometimes call these other files *header files* because they contain information that typically appears at the beginning, or head, of a program file. We also refer to these files as *include files*, for the obvious reason. In C language programs, the header (or include) files usually are recognizable by the `.h` extension to the file name; in assembler, the `.inc` extension is used. An include file may contain `#define` statements (C language) or `equate` statements (assembly language) that convey information to the file in which it is included.

Depending on the project, it may be more the rule than the exception that program modules use include files. To illustrate how this affects dependencies in a makefile, let's look again at the Borland version of the makefile we used at the beginning of this section:

```
# Define suffixes

.suffixes:
.suffixes:  .c  .obj  .exe

# Define macros

CFLAGS          =  -c -ms
OBJS            =  prog1.obj  prog2.obj
TARGET          =  prog.exe

# Target and Dependencies

$(TARGET):        $(OBJS)
        $(CC)  -e$@  $(OBJS)

prog1.obj:      prog1.c

prog2.obj:      prog2.c
```

(The lines with `prog1.c` and `prog2.c` ordinarily would not appear in the makefile because of `.suffixes:` and the predefined inference rule; we are showing them in this case to illustrate the need for dependency checking.)

Based on this makefile, the only dependency for `prog1.obj` is `prog1.c`. But suppose that `prog1.c` uses the include file `prog1.h`. The relevant line in the makefile

```
prog1.obj:  prog1.c
```

gives no indication that `prog1.h` has anything to do with this dependency.

Now let's say that we have changed `prog1.h` — but not `prog1.c` — since the last time our makefile produced the `.obj` file for `prog1.c`. Because `prog1.c` includes `prog1.h`, we want MAKE to recompile `prog1.c` when either it or `prog1.h` is modified. But MAKE does not know this. In this situation, MAKE does not know to check the date and time of `prog1.h`. It checks the date and time of `prog1.obj` and of `prog1.c` only and does not recompile `prog1.c` until `prog1.c` is modified.

An obvious solution to the apparent dilemma presents itself. We need simply to change our makefile to show that `prog1.obj` depends on both files:

```
prog1.obj:  prog1.c  prog1.h
```

It is clear now that `prog1.h` is a direct dependency for `prog1.obj`.

It is vital, as this example shows, that we state all the dependencies for every file we need to work with in the makefile.

The importance of this requirement cannot be overstated; however, it is not always easy to remember every include file in every program module. But because we are using the Borland version of the *make* utility, we are in luck!

One feature Borland provides is the availability of built-in dependency checking for all source files. Recall from page 98 that the `a` option instructs MAKE to perform dependency checking for us. When we use this option, the source programs are checked for include files and an internal dependency list is created for MAKE to use. The makefile itself does not change, unless we use the `K` option along with the `a` option. (Remember that the `K` option tells MAKE to keep the temporary files it creates; otherwise, they are automatically removed.)

Thus, to build a complete and accurate dependency list for our makefile, we specify the `a` option when we invoke MAKE:

```
make  -a
```

Because we issued the `make` command with the `a` option, MAKE determines whether any source files contain include files (and, if so, searches the current

directory and the directories listed in the INCLUDE environment variable), then builds the .obj targets accordingly.

Borland provides an alternative to using the a option on the command line. Instead we can insert in the makefile the statement:

```
.autodepend
```

This statement provides the same functionality as when we invoke MAKE with the a option as shown above. This technique offers the advantage of ensuring that built-in dependency checking is performed every time the makefile is executed. In contrast, if we rely on invoking the feature from the command line, we may inadvertently omit the a option when we issue the make command.

In either case, we see that the job of building the dependency list for any makefile is much easier to accomplish when we use the built-in dependency checking feature. This feature is so helpful, in fact, that we are well-advised to use it liberally, either by means of the a option on the command line or the .autodepend statement within the makefile.

If we want to generate the list of dependencies within the makefile, we can use the depends utility described in Appendix I. For an example, refer to page 82.

Input Files and In-Line Response Files

Remember the very first makefile we looked at in Chapter 1? Here it is again:

```
prog.exe:  prog1.obj  prog2.obj
        bcc  -eprog.exe  prog1.obj  prog2.obj

prog1.obj:  prog1.c
        bcc  -c  -w  -O2  prog1.c

prog2.obj:  prog2.c
        bcc  -c  -w  -O2   prog2.c
```

Notice that we linked the .obj files and produced the target prog.exe using the command line:

```
bcc  -eprog.exe  prog1.obj  prog2.obj
```

An alternative to using `bcc` to link is using the `tlink` command explicitly:

```
tlink  prog1.obj  prog2.obj, prog.exe ;
```

Notice the semicolon (;) at the end of the `tlink` command line. The purpose of the semicolon is to instruct the linker to use default values for the link options instead of prompting the user for them. If we want to use only some (or none) of the default values, we simply include on the command line the options we want before (or instead of) the semicolon.

The following command line shows a case in which we have decided not to use all the default options:

```
tlink  prog1.obj  prog2.obj,  prog.exe,  ,  cs ;
```

(The `tlink` command accepts the names of the `.obj` files separated by either blanks or plus signs (+).)

This `tlink` command line includes an option that was not present in the earlier case: `cs`, which tells the linker which library to bring in to link with our object files.

Notice that we retained the semicolon at the end of the command line. Without it, the linker might prompt for other options that we haven't specified. However, because the semicolon is present, the linker knows to use default values for the rest of the link options.

Our example `tlink` command line is pretty compact, but command lines can easily become quite lengthy. Command lines in makefiles can be particularly bothersome because *make* requires a backslash (\) at the end of a line to indicate that the next line is a continuation of the current line.

The solution to dealing with lengthy, unwieldy command lines in a makefile is simple. Using our example, we place the parameters to the `tlink` command in a file. Then, in the makefile, we tell the linker to use the file instead of depending on a list of parameters on the command line.

To get an idea of how this works, let's put our link parameters into a file called `makefile.lnk`. Here they are again:

```
prog1.obj  prog2.obj,  prog.exe,  ,  cs ;
```

We've just created an input file to the linker. (An input file is also referred to as a *response file* or an *autoresponse file.*) To have the linker read the input file, we precede the name of the input file with the "at" character (`@`). (Note that this use of `@` is distinct from the predefined macro `$@`.) This character tells the linker to use the information stored in the file and not to prompt for input. Now the `tlink` command line within the makefile looks like this:

```
tlink  @makefile.lnk
```

It's easy to see, even with our compact example, how helpful an input file can be. Its usefulness is even more evident when a large number of files and linker options must be used.

Now that we understand the basic concept of the input file to a linker, let's consider how we can make it even more useful. For example, what if we want to use a macro in an input file? We can do it: the linker is aware of environment variables.

Recall that in Chapter 4 we created an environment variable called `ROOTDIR` and showed how we could use it as if it were a macro. Here's an example of what our input file would look like using the environment variable as a macro:

```
prog1.obj  prog2.obj,  $(ROOTDIR)\prog.exe,  ,  cs ;
```

Closely related to input files are in-line response files. You can see the similarity in the following example:

```
tlink  @&&|
prog1.obj  prog2.obj,  $(ROOTDIR)\prog.exe,  ,  cs ;
|
```

In the earlier example, we told the linker (by means of the `@` character) to get its parameters from a file named `makefile.lnk`. In the example just above, we've inserted the notation `&&|` between the `@` character and the input to `tlink`. This means that `tlink` will get its input from a temporary file (indicated by `&&`) that contains the items beginning after the vertical line character (`|`) and continuing up to the next occurrence of the character. A file created in this manner is called an *in-line response file.*

Note that the in-line response delimiter can be any character that is not special to *make*. In our example we used the | character; however, we could have selected any character for our delimiter.

We can construct an in-line response with as many lines as we like. To do this, we simply add lines — without backslashes — and use the second occurrence of our delimiter (the | character in the example above) to indicate that we have completed the in-line response:

```
tlink  @&&|
prog1.obj  prog2.obj
$(ROOTDIR)\prog.exe
nul
cs ;
|
```

In each of these examples, MAKE generates a name for the temporary file it creates. The name of these files typically takes the form `MAKExxxx.$$$` (where `xxxx` is a four-digit number that MAKE generates). If we wish, we can use the `copy` command along with the in-line response function to create a file with a name that we choose. Here is how we can assign a specific name to this type of file:

```
copy  &&|
prog1.obj  prog2.obj,
$(ROOTDIR)\prog.exe
nul
cs ;
| makefile.lnk
```

In this example, the command copies the items between the delimiters (in this case, between | and |) to a file named `makefile.lnk`. We can use this file in the link statement as we did earlier:

```
tlink @makefile.lnk
```

Note that a file created by in-line response is a temporary file. After executing the makefile, MAKE removes any temporary files that were created in the process. This is true whether we assign a name or accept the generated file name.

But what if we want to save the in-line response file for some future use? If we want to keep an in-line response file, we must invoke MAKE with the

`K` option. This option prevents all temporary files that are created during execution of the makefile from being deleted when the makefile completes.

You can find out more about using input files and the in-line response function from the documentation provided with your compiler. Refer to the *Borland C++ Tools and Utilities Guide.*

Using Command Level Processing in Makefiles

In an ideal world, all programs would execute to completion successfully every time. Alas, this is not an ideal world. Despite our best efforts, the programs we develop do fail from time to time, for one reason or another. Thus, we use exit (or return) codes to determine whether a program failed and, often, what caused the failure.

In the DOS environment, the command processor supports the `errorlevel` statement. With this mechanism, we can test a program's exit code and take appropriate action based on the result. Consider the following simple example:

```
runxyz
if errorlevel 1 doprtscn
```

Here we execute the program `runxyz`, then test its exit code with the `errorlevel` statement. If `runxyz` returns an exit code of 1, we execute a program that writes the contents of the screen to the printer. If the exit code is any other value (or if `runxyz` does not return an exit code), no other action is performed.

This technique can be used effectively in batch, or command, processing. Unfortunately, it does not work for makefiles. MAKE does not provide any mechanism that allows for command level processing, such as testing a program's exit code. At least, not directly. But there is a way around this difficulty.

Let's say that we write a makefile in which we execute the program `runxyz`. The program fails. Now what?

Well, if we invoked MAKE without the `i` option, MAKE would stop when it received the exit code from `runxyz` indicating the program had failed. On the other hand, if we used the `i` option (or the `.ignore:` pseudotarget), MAKE would ignore the error reported by `runxyz` and continue processing the rest of the makefile.

Neither action suits our purpose. How then can we execute a program from within a makefile, determine whether the program worked correctly, and have MAKE take some alternative action based on the success or failure of the program?

The answer is to create a pseudotarget that uses the in-line response function we discussed in the previous section. With in-line response we can create a batch file that will perform the tasks we require: it will execute the program `runxyz`, test the exit code, and proceed as determined by the result of the test. Here is a segment of a makefile that illustrates this technique:

```
# Define pseudotargets, macros, etc.

      ⋮

# Define temp.bat as a dependency target

$(EXEC_NAME).exe: $(OBJS) temp.bat

      ⋮

# Execute the pseudotarget batch file (defined below)

   temp

      ⋮

# Define pseudotarget temp.bat --
# a batch file to perform command level processing

temp.bat:
   @copy Creating batch file  &&|
@echo off
runxyz
if errorlevel 1 doprtscn
| $@
```

Don't be confused by the various `@` characters you see in this example. As it happens, we use the same character to perform three separate functions.

In `@copy` and `@echo off`, the character performs its DOS shell function of suppressing the display of the function that is being called; in this case, `@` prevents each of those lines from being displayed. The second use of `@` is to signal the presence of an input file (in this case, the input file is an in-line response). We described this function of `@` on page 113. The third use of this character is in the predefined macro `$@`.

Now let's carry this example a step further. Suppose that `runxyz` can return either of two different error level codes, and we want to perform different operations depending on which error level the program returns. Before implementing this more complex solution, we must understand that error level values are cumulative in command processing. This means, for example, that if error level 2 is set, so is error level 1. With this in mind, here is the segment of a makefile that allows us to direct MAKE to take alternative actions based on the return code from `runxyz`:

```
temp.bat:
   @copy Creating batch file  &&|
@echo off
runxyz
if errorlevel 2 writeerr
if errorlevel 1 doprtscn
| $@
```

In this example, if `runxyz` returns error level 2, both `writeerr` and `doprtscn` are executed, in that order. (Note that we tested for error level 2 before error level 1.) If `runxyz` returns error level 1 only (that is, not error level 2), then only `doprtscn` is executed. If neither error level is set, no other action is performed.

By combining the in-line response function of MAKE with the concept of batch files, we can achieve the equivalent of DOS command level processing indirectly through our makefiles.

MAKE and the builtins.mak File

MAKE supports the use of a general file called `builtins.mak`. When we issue the `make` command on DOS, it automatically searches for this special file and, if it is present, includes the file as part of the makefile currently being executed. However, MAKE does not require `builtins.mak` to exist.

The `builtins.mak` feature provides a tool we can use to supplement the limited set of predefined macros that MAKE supports. We can also use this file to develop and maintain a set of inference rules (remember that MAKE has a limited set of predefined inference rules).

To make use of this feature, we create the file `builtins.mak`, which should reside in the same directory as `make.exe`, and include in it the macros and rules we want to use over and over again. We can illustrate this idea by making a few changes to our previous makefile (see page 105). We will remove the `include` directive and the inference rule and put them into `builtins.mak`. (Remember that once we remove the `include` directive, MAKE does not know about the `sample.mak` file or its contents.)

Here is our new makefile:

```
# Define suffixes

.suffixes:
.suffixes:  .c  .obj  .exe

# Define macros

.path.c   =  .
.path.obj =  .\sample2

OBJS = $(.path.obj)\prog1.obj $(.path.obj)\prog2.obj

# Test for defined macro

!if $d(DEBUG)
CFLAGS  =  $(CFLAGS)  -O-  -v
!endif

# Target and dependencies

$(EXEC_NAME).exe:  $(OBJS)
        $(CC)  -e$@  $(OBJS)
```

We see that this makefile still relies on the macros `EXEC_NAME` and `CFLAGS`; however, they are not defined in the makefile (we are assuming that the `DEBUG` macro was defined on the command line). Furthermore, we want MAKE to build `.obj` files from `.c` files using a specific path, but we

have not provided any rules for doing so. Nor have we told MAKE to include the contents of any other file that might contain this information.

How, then, does MAKE know what to do? Simply by reading the `builtins.mak` file, which we wrote to include the following lines:

```
CFLAGS      =  -c  -ms
EXEC_NAME   =  prog

.c.obj:
        $(CC) $(CFLAGS) -o$(.PATH.obj)\$*.obj $*.c
```

The technique of using `builtins.mak` works exactly like the `include` directive with one significant difference: the lines in `builtins.mak` will be used in every makefile executed by MAKE on our system (unless we use the `r` option), instead of only in those makefiles in which we remember to use the `include` directive.

Macro Precedence

The order of precedence for macros in Borland's MAKE is simple: the last definition of the macro that MAKE sees is the definition that takes effect. Here are some specific instances that show how this works:

If a macro has been defined in `builtins.mak`, then MAKE uses that definition. If the same macro is redefined in the makefile, the latest definition takes effect.

A macro can be defined on the command line using the `D`*identifier* option. If we redefine the same macro in the makefile, that is considered its latest definition; again, the latest definition takes effect.

If a macro is defined on the command line and is redefined in `builtins.mak`, the redefinition from `builtins.mak` takes effect.

Using MAKE to Develop GUI Applications

More and more applications are being developed for graphical user interfaces (GUIs) such as Windows from Microsoft, which runs on DOS systems. As we demonstrate in this section, programmers who use Windows libraries and resources to develop software can combine those tools with MAKE to create GUI applications.

In comparison to the basic DOS program development we've discussed so far, Windows-type development requires a different approach to creating the various objects and the final executable file. The Borland C++ Compiler and the Resource Workshop provide many of the libraries that we need to create a Windows application.

If we wish to develop a Windows application, we must have:

- Windows compiled program(s)
- Borland C++ Windows interface program — `C0Wx.obj`, where `x` is the memory model being used: `s` (small), `l` (large), and so on.
- Windows libraries – `CWx.lib`, where `x` is the memory model being used: `s` (small), `l` (large), and so on; `graphics.lib`, `import.lib`
- Any resource files (`.res` files) that are a part of the executable
- Module definition file

You can find detailed information about these items in the Borland C++ Version 3.1 reference manuals. Here we want to highlight just a few important points:

- The Borland C++ Windows interface program is an object file that you must link with your application for it to be Windows-compatible. This object file *must* be the first file given as input to the `tlink` command.
- You must use the Borland C++ libraries for Windows when you link your application files. The library files can be given as input to the `tlink` command in any order; however, all three must follow the Borland C++ interface program described previously.
- You must use the Microsoft Resource Compiler (RC) to define and compile any resource files that your application will use. The resulting resource file is appended to the end of the executable and is considered read-only by the program. (See Figure 5.2.) The resulting resource file can be created from multiple source files that represent different sections of the application.
- The linker uses the module definition file to determine what type of executable to create. The module definition file is the *only* way you

can tell the linker that you want to create a Windows application. Without a module definition file, the linker creates only non-Windows applications.

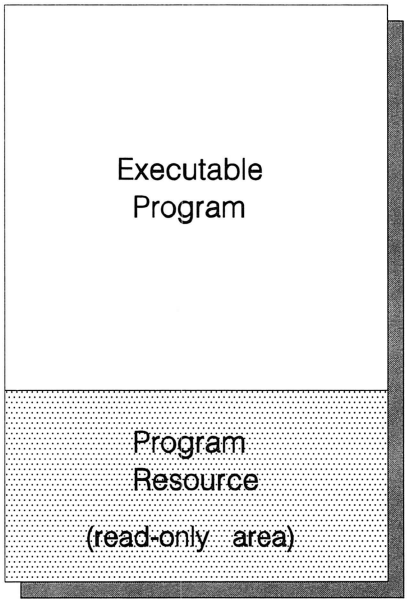


Figure 5.2. DOS GUI Format

Now that we have some understanding of the basic requirements, we are ready to look at an example. The following makefile (`winsamp.mak`) creates a window with text and two pull-down menus. One accesses the dialog feature, and the other exits the program. (This example demonstrates many of the *make* and MAKE functions we've already discussed, as well as the development of a GUI application.)

```
#
# Borland Makefile - Windows Example
#
# Mastering Make: A Guide to Building Programs on
# DOS, OS/2, and UNIX* Systems
# Tondo, Nathanson, Yount
#
# Prentice Hall (C) 1994
```

```
#

#
# Define suffixes
#

.suffixes:
.suffixes: .c .obj .exe .rc .res .dlg .def

#
# Define other makefile directives
#

.autodepend

.path.obj = .

#
# Define macros
#

CC      = bcc
CINC    = -I.
LINK    = tlink
LIBS    = cws graphics import
RC      = rc
RCFLAGS = -r
TARGET  = winsamp.exe

#
# Test for defined macro
#

!if !$d(DEBUG)
DEBUG=yes
!endif

#
# Define macros for compiling and linking
```

```
#

!if $(DEBUG) == "yes" || $(DEBUG) == "YES"
CFLAGS  = -c -R -ms -2 -v -d -W
LFLAGS  = -v -x -c -P- -Twe
!else
CFLAGS  = -c -R -ms -2 -d -W -O
LFLAGS  = -x -c -P- -Twe
!endif

#
# Define inference rules
#

.c.obj:
        $(CC) $(CFLAGS) $(CINC) $*.c

.rc.res:
        $(RC) $(RCFLAGS) $*.rc

#
# Define objects used to create the target
#

OBJS    = $(TARGET:.exe=.obj)

WINOBJ  = c0ws.obj

#
# Define the target to build
#

$(TARGET): $(OBJS) $(TARGET:.exe=.def)  \
                   $(TARGET:.exe=.res)
        $(LINK) $(LFLAGS) @&&|
$(WINOBJ)  $(OBJS)
$@
nul
$(LIBS)
```

```
$(TARGET:.exe=.def)
|
        $(RC) -t $(TARGET:.exe=.res) $@

#
# Define module definition file used to create executable
#

$(TARGET:.exe=.def):
        @copy &&|
NAME          WinSamp

DESCRIPTION   'Mastering Make Example - Borland'

EXETYPE       WINDOWS

CODE          PRELOAD MOVEABLE DISCARDABLE
DATA          PRELOAD MOVEABLE MULTIPLE

HEAPSIZE      1024
STACKSIZE     5120
| $@

#
# Define indirect dependencies
#

$(TARGET:.exe=.obj): $(TARGET:.exe=.c)

$(TARGET:.exe=.res): $(TARGET:.exe=.rc)  \
                     $(TARGET:.exe=.dlg)
```

This is certainly the most complex makefile we've seen so far. Let's take a closer look at its various parts.

We begin by clearing MAKE's built-in list of suffixes, then use the `.suffixes:` pseudotarget to tell MAKE what suffixes we want to use. Next we define directives that tell MAKE to check the dependency of all files named in the makefile and where the object files are to be created.

In the next section we define the various macros we plan to use in our makefile, such as the libraries to be linked in and the name of the target file. Because we are interested in whether the makefile will build a debug or production version of the program, we test to see if the macro `DEBUG` is defined. If `DEBUG` is not defined, then the definition of the macro becomes `yes`. We will use this macro to determine whether we are going to build a debug or production version of the program. The result of the test allows us to set the correct values for the other compile- and link-related macros.

Next, we want to make sure that MAKE knows how to create object (`.obj`) and resource (`.res`) files, so we define inference rules for them using some of the suffixes we defined at the beginning of the makefile. Finally, after defining the macro for the object files, we reach the target that we want to build. Notice that we include as dependencies not only the object file `winsamp.obj` (macro `OBJS`), but also the resource file `winsamp.res` (macro `$(TARGET:.exe=.res)`) and the module definition file `winsamp.def` (macro `$(TARGET:.exe=.def)`).

Now we come to the heart of this makefile. You may wonder why we have included the module definition file (`.def`) in the dependency list for an executable. Before we address that question, notice first that after defining the target, we use MAKE's in-line response feature to perform two separate and distinct functions. In the first instance, we use in-line response to provide information to the linker. (We saw this usage earlier in this chapter.) But the second instance — ah, this is what catches our attention. Here we use in-line response to define a pseudotarget that effectively causes MAKE to create the module definition file for us! This strategy assures us that two conditions — critical to our Windows application — are met:

- The module definition file is present.
- The module definition file is current.

Somewhat as an anticlimax, we conclude the makefile by defining the indirect dependencies for the object file `winsamp.obj` and the resource file `winsamp.res`.

Notice that the `$(TARGET:.exe=.dlg)` file is a part of the direct dependency for `$(TARGET:.exe=.res)`. This is because when the resource file is compiled, it includes all dialog files (`.dlg`) as part of the source file. Refer to the Windows sample source file (`.rc`) to see how the include process occurs. We must define the dialog file as we would any include file for a source file.

As we see from this example, no mystery attaches to using MAKE to create a GUI application. The makefiles we write to build Windows applications are very much like any other makefile. We use the same fundamentals and similar techniques. As in any development project, we need only take care that our program conform to the requirements of the environment.

The Windows example we have just discussed presents a single point of view of how best to use MAKE in a Windows development effort. Refer to Borland's user documentation for more information about Windows development and C++ applications. The complete source code for this example appears in Appendix F.

Summary

Two *make* utilities available on DOS are NMAKE from Microsoft and MAKE from Borland. Each version of *make* supports an extensive list of command line options. A number of these options are unique to NMAKE; others are unique to MAKE.

- Refer to page 68 for a list of NMAKE options.
- Refer to page 98 for a list of MAKE options.

Both versions of *make* also support preprocessing directives. These directives allow us to control the operation of *make* with conditional and other special-purpose statements. A preprocessing directive begins with ! in the first column of the line on which it appears. The utility processes any directives it finds in the makefile before executing the commands associated with targets.

For NMAKE:

- Two frequently used directives are `ifdef` and `if`. Any statement(s) associated with `ifdef` and `if` must be followed by an `endif` directive. Both directives can also use the `else` directive to establish alternative actions for NMAKE to take.
- The `ifdef` directive determines whether the name that follows it has been defined for the current makefile. The `if` directive evaluates the

expression that follows it (`if` *expr*) and executes the statements until the `endif` or `else` when *expr* is nonzero.

- A special use of the `if` directive is to pass a command directly to `command.com`. In this case, the command to be passed must be enclosed in `[ ]`.

For MAKE:

- One frequently used directive is the `if` directive. Any statement(s) associated with this directive must be followed by an `endif` directive. The `if` directive can also use both the `elif` and the `else` directives to establish alternative actions for MAKE to take.

- When followed by the name of a macro, the `if` directive tests the value assigned to that macro. For example:

```
if $(DEBUG) == 3
```

- The predefined macro `$d` can be used with the `if` directive to determine whether that macro has been defined for the current makefile; its use can be extended with the NOT operator (`!`) to test that a macro has not been defined. For example:

```
if $d(DEBUG)
```

Both versions of *make* support the same conditional operators as the C programming language. When we use these operators with the utilities' respective conditional preprocessing directives, we can extend the power of *make* to achieve complex testing and branching activities from within the makefile.

Both versions of *make* recognize the \ character as a line continuation character. However, we may want *make* to see \ as a literal character; for example, to represent the root directory on DOS. Each version of *make* has its own strategy for dealing with this situation.

For NMAKE:

- The caret (^) is a special control character that turns off the special meaning of other characters. When used before a backslash (^\), it tells NMAKE that the backslash is a literal character and not a line continuation indicator.

- Similarly, when used in a macro definition before a newline character, the ^ tells NMAKE that the newline is part of the definition, not an

end-of-line marker for NMAKE. Another way to represent the newline character without having to use ^ is to use `\n` where the newline should occur.

For MAKE:

- Because the utility does not support escape sequences (such as ^), we use \\ to represent a literal \ character.

A complete and correct list of dependencies is critical to the successful operation of *make*. Source files that *make* uses to build a target often contain header files (`.h`) or include files (`.inc`). These can easily be omitted from a makefile, usually with unwanted (if not disastrous) results.

For NMAKE and MAKE:

- Dependency checking can be performed dynamically as *make* executes to ensure that all dependencies are known. Refer to Appendix I for an example of a program that produces an accurate, up-to-date list of dependencies for a makefile.

For MAKE only:

- The `a` option issued on the command line allows dependency checking to be performed dynamically as MAKE executes to ensure that all dependencies are known to MAKE. An alternative technique is to use the `.autodepend` statement within the makefile to ensure dependency checking occurs every time the makefile is executed.

An input file provides input to a command; in this case, to a command within a makefile. In-line response files are closely related to input files. The difference is that we create an input file external to and independent of the makefile in which it is used. In contrast, we write an in-line response file as part of the makefile, and the file is created as *make* executes. In either case, we use the "at" character (`@`) to tell a linker to use an input file.

For NMAKE:

- The delimiters for in-line response are `<<...<<`.
- We assign a name to the temporary file that is created by in-line response. If a file of the same name already exists, it is overwritten.
- If we want to prevent NMAKE from removing the in-line response file, we must follow the closing << delimiter with the `keep` command.

For MAKE:

- The `&&` notation indicates the beginning of the temporary file created by in-line response. The delimiters for in-line response can be any character that is not special to *make* (for example `&&|...|`).
- The file created by in-line response is assigned a name that is generated automatically; optionally, we can assign a name of our choice. If a file of the same name already exists, it is overwritten.
- If we want to prevent MAKE from removing the in-line response file, we must invoke `make` on the command line with the `K` option.

By combining the in-line response function of *make* with the concept of batch files, we can achieve the equivalent of DOS command level processing from within a makefile.

Both Microsoft and Borland support a special file where we can store macros and inference rules that we define and want to use frequently.

- For NMAKE, the special file is called `tools.ini` file.
- For MAKE, the special file is called `builtins.mak`.

Each version of *make* reads the respective special file automatically (unless we use the `r` option). Thus, these files provide a convenient device for customizing default options for *make*.

For each version of *make*, if a makefile and the special file contain conflicting definitions or rules, *make* uses the makefile in preference to the special file. In this respect, both versions of *make* follow the same precedence. However, in other respects the order of precedence varies for the two versions.

- For NMAKE the precedence of macros from highest (1) to lowest (5) is:
  1. macros on the command line
  2. macros in the makefile
  3. inherited macros (environment variables)
  4. macros in the `tools.ini` file
  5. predefined macros

When we use the `e` option we tell NMAKE that environment variables take precedence over macro names that appear in a makefile. In this case, the order of precedence for macros (highest (1) to lowest (5)) is:

1. macros on the command line
2. inherited macros (environment variables)
3. macros in the makefile
4. macros in the `tools.ini` file
5. predefined macros

- For MAKE the precedence of macros from highest (1) to lowest (4) is:

  1. macros in the makefile
  2. macros in the `builtins.mak` file
  3. inherited macros (environment variables)
  4. macros on the command line

  When we use the `e` option we tell MAKE that environment variables take precedence over macro names that appear in a makefile. In this case, the order of precedence for macros (highest (1) to lowest (4)) is:

  1. macros in the makefile
  2. inherited macros (environment variables)
  3. macros in the `builtins.mak` file
  4. macros on the command line

We can combine Windows tools with *make* to create graphical user interface (GUI) applications. The Windows tools we need are Windows-compiled programs; Windows libraries; a Resource Compiler and resource files; a module definition file; and, for MAKE, a Windows interface program. The in-line response function of *make* allows us to create the module definition file from within the makefile so we are assured that the module definition file is always present and always current.

Chapter 6

The MAKE Utility on OS/2 Systems

In earlier chapters we looked at concepts and strategies that, for the most part, apply universally to all versions of *make*. Here we focus our attention on the *make* utility on OS/2 systems; specifically, NMAKE from IBM and MAKE from Borland.

We have structured this chapter on two premises. First, that you are interested in how *make* works on an OS/2 system and, second, that you have focused on either IBM's NMAKE or Borland's MAKE. Thus, the chapter is organized in two essentially parallel sections — one for each version of the *make* utility on OS/2 — and a concluding summary that covers both versions. If you read both the NMAKE section and the MAKE section, you will notice some duplication of material.

NMAKE on OS/2

We begin this section by showing how to invoke NMAKE. Then we explore how some of the topics we met earlier — such as options, dependencies, continuation lines, and environment variables — apply specifically to IBM's NMAKE. New topics introduced in this section include preprocessor directives, input files and in-line response files, and the `tools.ini` file. We conclude with some tips for using NMAKE to develop applications for a graphical user interface (GUI) environment.

Invoking NMAKE

Here is the format for invoking NMAKE on the command line:

```
nmake [options] [macro_definitions] [targets]
```

The brackets ([ ]) around a field name indicate that the field is optional. Thus, all three fields are optional for NMAKE.

Command Line Options

NMAKE supports an extensive list of command line options. If you are familiar with other versions of *make*, some of these options may look familiar. However, a number of NMAKE options differ from those supported in other versions of *make*.

NMAKE is not case sensitive with respect to options; that is, NMAKE treats `a` and `A` as the same option. Remember that when we specify an option on the command line, we must precede the option letter with a dash (`-`) or a forward slash (`/`).

Here are the options that NMAKE supports:

`?`	Display short help
`a`	Build all targets
`c`	Suppress messages
`d`	Display modification dates
`e`	Let environment variables override macro definitions
`f` *filename*	Specify makefile to be *filename*
`help`	Display help information
`i`	Ignore exit/error codes
`n`	Display commands but do not execute them
`nologo`	Suppress copyright message
`p`	Print values of defined macros and pseudotargets
`q`	Return exit code

`r`	Ignore `tools.ini` file
`s`	Suppress command display while executing
`t`	Change target modification dates
`x` *filename*	Specify destination for error messages

Preprocessing Directives

A preprocessing directive is a command or an instruction that allows us to conditionally execute other commands or to conditionally select parts of a makefile. When NMAKE reads a makefile, it processes any preprocessing directives it finds before it executes commands associated with targets.

A preprocessing directive begins with an exclamation mark (`!`) and must appear at the beginning of the line. That is, we must place the `!` in the first column on the line. However, after the `!` we can use blank spaces and/or tabs, if we wish, to enhance readability of the file.

Here are the preprocessing directives that NMAKE supports:

`cmdswitches`	Change options in NMAKE
`error` *message*	Display error message
`include <`*filename*`>`	Insert another file as part of this makefile; NMAKE searches for the file in the directories specified in the environment variable `INCLUDE`
`include "`*filename*`"`	Insert another file as part of this makefile; NMAKE searches for the file in the current directory
`if` *expr*	Execute statements following `if` when *expr* evaluates to nonzero
`ifdef` *name*	Execute statements following `ifdef` when *name* has been defined as a macro
`ifndef` *name*	Execute statements following `ifndef` when *name* has not been defined
`else`	Execute statements following `else` when *expr* in the preceding `if`, or *name* in the preceding `ifdef` or `ifndef` evaluates to zero

`endif`	Mark the end of an `if`, `ifdef`, or `ifndef` directive
`undef` *name*	Undefine *name*

We can use conditional operators with the `if` and `else` preprocessing directives. The conditional operators that NMAKE supports are the same as those supported in the C programming language. They are:

- Unary operators
  - `-` arithmetic negation
  - `~` bitwise complement
  - `!` logical NOT
- Binary operators

`+`	addition	`>>`	right shift
`-`	subtraction	`<<`	left shift
`*`	multiplication	`&`	bitwise AND
`/`	division	`|`	bitwise OR
`%`	modulus	`^`	bitwise XOR
`&&`	logical AND	`>=`	greater than or equal
`||`	logical OR	`<=`	less than or equal
`>`	greater than	`==`	equality
`<`	less than	`!=`	inequality

We can use four of the binary operators listed above to compare string expressions as well as numeric expressions. The operators that we can use for string comparison are `>=`, `<=`, `==`, and `!=`.

To see how we can apply preprocessing directives, let's start with a makefile from Chapter 3. Here is the makefile:

```
CFLAGS =  -W2  -O
OBJS   =  prog1.obj  prog2.obj
TARGET =  prog.exe

$(TARGET):  $(OBJS)
        $(CC)  -o  $@  $(OBJS)
```

Now let's modify this makefile to use directives. Here is what we want to achieve:

- Change the compiler options to conform to IBM C Set/2++ compiler.

- Include another makefile (called `sample.mak`) as part of our makefile. The macros we want to use will be defined in this file.
- If the `DEBUG` macro has been defined, then add the `Ti+` option to `CFLAGS` (`Ti+` turns source debugging on).

Here is the makefile in its new form:

```
# Define suffixes

.SUFFIXES:
.SUFFIXES: .c .obj .exe

# Include the file sample.mak

!include  "sample.mak"

# Test for defined macro

!ifdef DEBUG
CFLAGS  =  $(CFLAGS)  -Ti+
!endif

# Target and dependencies

$(EXEC_NAME).exe:  $(OBJS)
        $(CC) -Fe  $@  $(OBJS)
```

Let's look more closely at the new features of this makefile, beginning with the `include` directive. We have used the `include` directive

```
!include  "sample.mak"
```

to tell NMAKE to read the file `sample.mak` from the current directory into the current makefile. When NMAKE begins to process the makefile, it appears as if the contents of `sample.mak` were always present. Effectively, they are: the directive brings them in. Here are the contents of `sample.mak`:

```
# Define macros

CFLAGS          =  -c  -W3
OBJS            =  prog1.obj  prog2.obj
EXEC_NAME       =  prog
```

By using the `include` directive, we can easily create part of a makefile — a sequence of statements — that is common to various makefiles we use. Then, whenever we need this particular sequence of statements, we tell NMAKE to include the file that contains them.

Let's look now at the `ifdef` directive. Remember that this directive asks NMAKE whether a particular macro has been defined. For our immediate interest, the directive

```
!ifdef DEBUG
```

asks NMAKE if a macro named `DEBUG` has been defined.

We can define the macro `DEBUG` in the makefile:

```
DEBUG  =  YES
```

Or we can define the macro on the command line when we invoke NMAKE:

```
nmake  DEBUG=YES
```

If `DEBUG` is defined, NMAKE executes the statements between the `ifdef` and the `endif` directives. In our case, NMAKE would change `CFLAGS` to be defined as:

```
CFLAGS  =  -c  -W3  -Ti+
```

It is important to note that `CFLAGS` is left justified on the line; that is, the `C` of `CFLAGS` appears below the `!`. We must begin `CFLAGS` in the first column of the line because it is a macro name, not a command. Remember that only commands are indented on the line (with one or more spaces or tabs).

If `DEBUG` has not been defined, then NMAKE does not execute the statements between the `ifdef` and the `endif` directives; `CFLAGS` remains as originally defined.

Let's consider using the `DEBUG` macro in a different way. With the `if` and the `endif` directives, we can explicitly test whether `DEBUG` has been defined to be `YES` (or any other value we want to test for):

```
!if "$(DEBUG)" == "YES"

      ⋮

!endif
```

We can use an `else` directive following the `if` directive:

```
!if "$(DEBUG)" == "YES"

      :

!else

      :

!endif
```

This variation means that when `DEBUG` is defined to be `YES`, NMAKE executes the statements between the `if` and the `else` directives. When `DEBUG` is defined to be some value other than `YES` (or is not defined at all), NMAKE executes the statements between the `else` and the `endif` directives.

Let's add our debug flags to `CFLAGS` in the first case; let's keep `CFLAGS` simple in the second case:

```
!if "$(DEBUG)" == "YES"
CFLAGS =  -c  -W3  -Ti+
!else
CFLAGS =  -c  -W3
!endif
```

Notice in these examples of the `if` directive that we use the notation `$( )` around `DEBUG` and we use double quotation marks on both sides of the expression. (Contrast this notation with the earlier example of the `ifdef` directive where we used neither. We will return to this distinction in a minute.)

We must use `$( )` around `DEBUG` because we want the `if` directive to evaluate the *value* of `DEBUG`. Without `$( )`, `if` would compare the literal string `"DEBUG"` with the string `"YES"`. The test, of course, fails — and is not what we intended in any event.

We must use the quotation marks because we have asked NMAKE to evaluate a comparison, and the values to be compared must be of the same type. In our case, the value we want to test, `YES`, is a string; this means that the other value, `$(DEBUG)`, must also be a string. We tell `if` to evaluate the comparison of two strings by surrounding each value with string delimiters (i.e., quotation marks). Without the quotation marks, `if` would expect to compare two integers.

Now let's consider the syntax of the `ifdef` directive. Recall that we used neither the notation `$( )` nor quotation marks around `DEBUG`. This is because `ifdef` determines only whether the macro name has been defined; it does not check for the value of the macro, nor does it require the macro name to be a particular type.

We can write more than one statement following an `if` or an `else` directive. For example, we may need to change the OS/2 environment variable `INCLUDE` to add a directory to the list of directories that the C preprocessor searches.

In our case, we want to add the `\DEBUG` directory to the `INCLUDE` environment variable when the macro `DEBUG` is assigned the value `YES`. Here are the lines we need in our makefile:

```
!if "$(DEBUG)" == "YES"
CFLAGS = -c -W3 -Ti+
!       if [set INCLUDE=%INCLUDE%;\DEBUG]
!       endif
!else
CFLAGS = -c -W3
!endif
```

Recall from Chapter 4 that we use the OS/2 `set` command to change the value of an environment variable. For NMAKE to perform this operation, the command must be within an `if` directive, where it always evaluates as nonzero. The brackets around the command are required, and the information within them

```
set INCLUDE=%INCLUDE%;\DEBUG
```

is passed directly to `cmd.exe`.

We conclude this `if` directive right away with an `endif` because we do not have any other statements to execute after the `set` command.

From this brief discussion of preprocessing directives, we've gained some insight into their usefulness in makefiles. Different circumstances will suggest other uses. For more information about this versatile feature, refer to the online help that IBM provides with NMAKE.

Expressions in Preprocessing

IBM's NMAKE supports the use of C-like expressions in the preprocessing of a makefile. This feature allows us to create complex expressions that will

determine which of several alternative actions the makefile will perform.

Let's explore how we can use complex expressions in a makefile. For this example, here is what we want to achieve:

- Place the executable file in its own directory, which we will call `execute`. The directory name will be defined as a macro in this makefile.
- Determine the version type of the program being created. If we are creating a debug version of the program, we want to copy the source code into the directory `execute`, which will allow debugging. If we are creating a non-debug (or production) version, then we want to delete all source code programs from that directory.

Here is the makefile that will accomplish these objectives:

```
# Define suffixes

.SUFFIXES:
.SUFFIXES: .c .obj .exe

# Include the file sample.mak

!include  "sample.mak"

# Define macros

DIRECTORY = ..\execute

# Test for defined macro

!ifdef DEBUG
CFLAGS          =  $(CFLAGS)  -Ti+
!endif

# Check to see if directory exists; create if it does not

!if !exist ($(DIRECTORY)) && !defined (DEBUG)
!   if [md $(DIRECTORY)]
!   endif
```

```
!endif

# Check to see what version we are creating

!ifdef DEBUG
!   message Copying source files
!   if [copy *.c $(DIRECTORY)]
!   endif
!else
!   if [if exist $(DIRECTORY)\prog1.c del *.c]
!   endif
!   message Deleting source code files
!endif

# Target and dependencies

$(DIRECTORY)\$(EXEC_NAME).exe:  $(OBJS)
        $(CC) -Fe  $@  $(OBJS)
```

Now let's examine this makefile in some detail. We define the macro `DIRECTORY` to identify the output directory where we want to store the executable file (and the source code for debug versions). After defining the macros, we check for the existence of the output directory; to do this, we use the `if exist` directive to form a Boolean expression.

Because `if exist` *path* is a Boolean expression, it evaluates to true or false.

```
!if !exist ($(DIRECTORY)) && !defined (DEBUG)
!   if [md $(DIRECTORY)]
!   endif
!endif
```

In our case, we want to perform an action if the output directory does not exist and the macro `DEBUG` is not defined, so we use the negation expression (`!`), which is not to be confused with the `!` that identifies a preprocessor directive in the makefile. If the negated expression evaluates true (that is, if the directory does not exist and `DEBUG` has not been defined), we create it with the DOS shell command `md`.

Next we determine the kind of executable file we are creating. If it is a debug version, then we want to copy the current source code files to

`$(DIRECTORY)`. On the other hand, if we are creating a production version of the program, then we do not want the source code in that directory and we want to remove all source files (`*.c`) that are there. Let's look a little more closely at this section of the makefile.

```
!ifdef DEBUG
!   message Copying source files
!   if [copy *.c $(DIRECTORY)]
!   endif
!else
!   if [if exist $(DIRECTORY)\prog1.c del *.c]
!   endif
!   message Deleting source code files
!endif
```

First, we use the `ifdef` directive to determine whether a macro named `DEBUG` has been defined (perhaps on the command line where NMAKE was invoked). If the macro has been defined, we use the `message` directive to report the action we are going to take, which is to copy the source files to the output directory. However, if the macro has not been defined, we follow up with a complex expression that determines whether any source files exist in the output directory. If so, we remove the files and use the `message` directive again to report that action.

As this example shows, by judiciously applying preprocessor directives and complex expressions, we can make decisions within a makefile that determine the course of action the makefile will take. These techniques expand the power of *make*, helping us to accomplish with relative ease tasks that would otherwise be much more troublesome.

Special Cases

We know from earlier chapters that when we construct and modify makefiles, we may need more than one line to specify a list of dependencies or a command with its various parameters. NMAKE accepts a backslash (\) before the end of a line as a signal that the current line has not ended. This means that the next line is the continuation of the line that ends with \.

This convention is helpful with long lines. This convention can also present a problem. For example, what will NMAKE do if it sees the lines

```
cd  \
copy  autoexec.bat  autoexec.old
```

in a makefile? The first command tells NMAKE to change to the root directory (in OS/2, the \ character is the root directory for a device). Notice that this command ends with \, which is the last item on the line. Because \ means the root directory in OS/2 and NMAKE uses \ as a line continuation indicator, we have a problem. Specifically, NMAKE understands the command to be

```
cd  copy  autoexec.bat  autoexec.old
```

which is scarcely what we had in mind!

Fortunately, we have a way to tell NMAKE when a \ is not meant to be a continuation indicator. We place a caret (^) in front of the backslash

```
cd  ^\
copy  autoexec.bat  autoexec.old
```

and NMAKE sees two command lines, as we intended.

The ^ turns off the special meaning — continuation indicator — for the backslash. This function of ^ extends to other characters in NMAKE. For example, the newline character signals the end of a line. If we want to use the newline as part of a string, we can turn off the meaning of newline by using a caret:

```
STRING=All^
is well.
```

We've defined the macro `STRING` to have two physical lines. Because ^ precedes the newline, `STRING` contains a newline after '`All`'. When we display the contents of `STRING`, we'll see '`All`' on one line followed by '`is well.`' on the next line. If we do not use the ^ in front of the newline, NMAKE will understand the value of `STRING` to be '`All`'. However, it will not recognize '`is well.`' as anything meaningful and consequently will display an error message.

Alternatively, we can insert a newline in a string by specifying the newline character as `\n`. So the two physical lines we defined previously for `STRING` become one line in the macro definition:

```
STRING=All\nis well.
```

In this case, of course, we do not need to use the ^ to turn off any special meanings. When we display the contents of `STRING` using this definition, we see two lines as we did before.

Creating Dependencies Dynamically

In Chapter 2 we mentioned the need for performing dependency checking. We are ready now to explore that concept further.

We well know by now that an important use of NMAKE is to maintain program modules within a development project. We also recall (from Chapter 2) that a source file may include other files.

We sometimes call these other files *header files* because they contain information that typically appears at the beginning, or head, of a program file. We also refer to these files as *include files*, for the obvious reason. In C language programs, the header (or include) files usually are recognizable by the `.h` extension to the file name; in assembler, the `.inc` extension is used. An include file may contain `#define` statements (C language) or `equate` statements (assembly language) that convey information to the file in which it is included.

Depending on the project, it may be more the rule than the exception that program modules use include files. To illustrate how this affects dependencies in a makefile, let's look again at the makefile we used in Chapter 3:

```
CFLAGS =  -W3  -O+
OBJS   =  prog1.obj  prog2.obj
TARGET =  prog.exe

$(TARGET):  $(OBJS)
        $(CC)  -Fe  $@  $(OBJS)

prog1.obj:  prog1.c

prog2.obj:  prog2.c
```

(The lines with `prog1.c` and `prog2.c` ordinarily would not appear in the makefile because of `.SUFFIXES:` and the predefined inference rules; we are showing them in this case to illustrate the need for dependency checking.)

Based on this makefile, the only dependency for `prog1.obj` is `prog1.c`. But suppose that `prog1.c` uses the include file `prog1.h`. The relevant line in the makefile

```
prog1.obj:  prog1.c
```

gives no indication that `prog1.h` has anything to do with this dependency.

Now let's say that we have changed `prog1.h` — but not `prog1.c` — since the last time our makefile produced the `.obj` file for `prog1.c`. Because `prog1.c` includes `prog1.h`, we want NMAKE to recompile `prog1.c` when either it or `prog1.h` is modified. But NMAKE does not know this. In this situation, NMAKE does not know to check the date and time of `prog1.h`. It checks the date and time of `prog1.obj` and of `prog1.c` only and does not recompile `prog1.c` until `prog1.c` is modified.

An obvious solution to the apparent dilemma presents itself. We need simply to change our makefile to show that `prog1.obj` depends on both files:

```
prog1.obj:  prog1.c  prog1.h
```

It is clear now that `prog1.h` is a direct dependency for `prog1.obj`.

It is vital, as this example shows, that we state all the dependencies for every file we need to work with in the makefile.

The importance of this requirement cannot be overstated; however, it is not always easy to remember every include file in every program module. But consider this: what if we devise some way to have NMAKE help us to satisfy the requirement?

The tool we have in mind is a program that creates a correct — and complete — list of dependencies for the makefile. Still working with the example from Chapter 3, our program will have to read the two identified indirect dependencies, `prog1.c` and `prog2.c`, and look for include files within them. The output of the program will be the original makefile plus a dependency list for each `.obj` file. Then any time we modify either `.c` file, we will rerun the program to create an updated dependency list.

We have, in fact, developed such a tool: the program `depends` in Appendix I builds a dependency list from the dependencies stated in the makefile. The program reads the current makefile to collect the object file names (`prog1.obj` and `prog2.obj`). `depends` then reads each source program (it looks for `.c` and `.asm` files for a given `.obj` target), looking for `include` statements, and updates the makefile to reflect the current dependency list.

We must make some changes to our makefile if we want to use `depends` to accomplish automatic dependency checking. Here is the modified makefile:

```
CFLAGS  =  -c  -W3  -O+

prog.exe:  prog1.obj  prog2.obj
        $(CC)  -Fe  $@  prog1.obj  prog2.obj

dependen:
        copy  makefile  makefile.old
        depends  -I.  -I%%ROOTDIR%%  >makefile
        $(MAKE)
```

Let's investigate the changes we made. First, we removed the dependency lines

```
prog1.obj:  prog1.c
prog2.obj:  prog2.c
```

because `depends` will generate them. We also created a pseudotarget, `dependen`, with three command lines:

```
copy  makefile  makefile.old
depends  -I.  -I%%ROOTDIR%%  >makefile
$(MAKE)
```

The `copy` command makes a backup copy of `makefile`. We save a copy of the makefile in `makefile.old` just in case something does not work the way we expect it to work. (Another reason is that `depends` reads the input from a default file named `makefile.old`. See Appendix I for information about naming the input file.)

The `depends` command executes the `depends` program using a few options. (The options are explained in Appendix I.) We redirect the output from `depends` (the original makefile and the dependency list) to the current makefile; that is, we overwrite the existing makefile. Finally, because the predefined macro `MAKE` expands to the `nmake` command, NMAKE is invoked recursively.

One note about the options for `depends`: we use `%%` around the environment variable `ROOTDIR` because `depends` expects to find an environment variable surrounded by %s, as in `%ROOTDIR%`. To NMAKE, `%` is a special character, so we must use two %s to mean one %. Using `-I%%ROOTDIR%%` allows us to pass `-I%ROOTDIR%` to `depends`.

Now, to build the dependency list, we invoke `nmake` on the command line and specify the target:

```
nmake  dependen
```

Here is the makefile after `depends` modifies it:

```
CFLAGS  =  -c  -W3  -O+

prog.exe:  prog1.obj  prog2.obj
        $(CC)  -Fe  $@  prog1.obj  prog2.obj

dependen:
        copy  makefile  makefile.old
        depends  -I.  -I%%ROOTDIR%%  >makefile
        $(MAKE)

prog1.obj: prog1.c      \
                f:\ibmcpp\include\stdio.h       \
                .\prog1.h

prog2.obj: prog2.c      \
                f:\ibmcpp\include\stdio.h
```

`depends` determined that `prog1.c` includes two files, `stdio.h` and `prog1.h`, and that `prog2.c` includes only one, `stdio.h`.

We can see that the job of building the dependency list for a makefile is much easier when we use a program like `depends`.

Let's review how this works. From the command line we execute

```
nmake  dependen
```

which causes NMAKE to begin executing the makefile at the pseudotarget `dependen`. NMAKE then makes a backup copy of the makefile, creates the dependency list, and writes it to `makefile`. When NMAKE reaches the third command line for the pseudotarget, it reinvokes itself and begins executing the same makefile from the top. This second instance of NMAKE finds `prog.exe` to be the first target in the makefile, so it begins to do all the things necessary to build `prog.exe`. Note that in this instance NMAKE does not execute the commands associated with the pseudotarget because `dependen` is not required for `prog.exe`. NMAKE does, however, find the dependency lines for `prog1.obj` and `prog2.obj` because `depends` inserted them on the first instance of NMAKE.

NMAKE does not find `dependen` the second time because that target is not part of `prog.exe` and we do not explicitly refer to that pseudotarget. In this case, the only way we reach `dependen:` is by using the target name on the command line.

Input Files and In-Line Response Files

Remember the very first makefile we looked at in Chapter 1? Here it is again, modified for NMAKE on OS/2:

```
prog.exe:  prog1.obj  prog2.obj
        icc  -Fe  prog.exe  prog1.obj  prog2.obj

prog1.obj:  prog1.c
        icc  -c  -W3  -G3  -O+  prog1.c

prog2.obj:  prog2.c
        icc  -c  -W3  -G3  -O+  prog2.c
```

Notice that we linked the `.obj` files and produced the target `prog.exe` using the command line:

```
icc  -Fe prog.exe  prog1.obj  prog2.obj
```

An alternative to using `icc` to link is using the `link386` command explicitly:

```
link386  prog1.obj  prog2.obj, prog.exe ;
```

Notice the semicolon (`;`) at the end of the `link386` command line. The purpose of the semicolon is to instruct the linker to use default values for the link options instead of prompting the user for them. If we want to use only some (or none) of the default values, we simply include on the command line the options we want before (or instead of) the semicolon.

The following command line shows a case in which we have decided not to use all the default options:

```
link386  prog1.obj  prog2.obj,  prog.exe,  ,  os2386 ;
```

(The `link386` command accepts the names of the `.obj` files separated by either blanks or plus signs (`+`).)

This `link386` command line includes an option that was not present in the earlier case: `os2386`, which tells the linker which library to bring in to link with our object files.

Notice that we retained the semicolon at the end of the command line. Without it, the linker might prompt for other options that we haven't specified. However, because the semicolon is present, the linker knows to use default values for the rest of the link options.

Our example `link386` command line is pretty compact, but command lines can easily become quite lengthy. Command lines in makefiles can be particularly bothersome because *make* requires a backslash (\) at the end of a line to indicate that the next line is a continuation of the current line.

The solution to dealing with lengthy, unwieldy command lines in a makefile is simple. Using our example, we place the parameters to the `link386` command in a file. Then, in the makefile, we tell the linker to use the file instead of depending on a list of parameters on the command line.

To get an idea of how this works, let's put our link parameters into a file called `makefile.lnk`. Here they are again:

```
prog1.obj  prog2.obj,  prog.exe,  ,  os2386 ;
```

We've just created an input file to the linker. (An input file is also referred to as a *response file* or an *autoresponse file*.) To have the linker read the input file, we precede the name of the input file with the "at" character (`@`). (Note that this use of `@` is distinct from the predefined macro `$@`.) This character tells the linker to use the information stored in the file and not to prompt for input. Now the `link386` command line within the makefile looks like this:

```
link386  @makefile.lnk
```

It's easy to see, even with our compact example, how helpful an input file can be. Its usefulness is even more evident when a large number of files and linker options must be used.

Now that we understand the basic concept of the input file to a linker, let's consider how we can make it even more useful. For example, what if we want to use a macro in an input file? We can do it: the linker is aware of environment variables.

Recall that in Chapter 4 we created an environment variable called ROOTDIR and showed how we could use it as if it were a macro. Here's an example of what our input file would look like using the environment variable as a macro:

```
prog1.obj  prog2.obj,  $(ROOTDIR)\prog.exe,  ,  os2386 ;
```

Closely related to input files are in-line response files. You can see the similarity in the following example:

```
link  @<<makefile.lnk
prog1.obj  prog2.obj,  $(ROOTDIR)\prog.exe,  ,  os2386 ;
<<keep
```

In the earlier example, we told the linker (by means of the @ character) to get its parameters from a file named makefile.lnk. In the example just above, we've inserted the notation << between the @ character and the input file makefile.lnk. This format indicates that the file makefile.lnk will be created from the items that follow, beginning on the next line and continuing up to the next occurrence of <<. A file created in this manner is called an in-line response file. If a file of the same name already exists (makefile.lnk in our example), it is overwritten by the input to the in-line response.

We can construct an in-line response with as many lines as we like. To do this, we simply add lines – without backslashes – and use the second occurrence of << to indicate that we have completed the in-line response:

```
link386  @<<makefile.lnk
prog1.obj  prog2.obj
$(ROOTDIR)\prog.exe
nul
os2386 ;
<<keep
```

Note that a file created by in-line response is a temporary file. After executing the makefile, NMAKE removes any temporary files that were created in the process. Now, we might want to save makefile.lnk for some future use. If we do, we must follow the << at the end of the in-line response with the command keep. If we are not interested in saving the file to use again, we omit keep. (The default is <<nokeep, which is equivalent to <<.)

Using Command Level Processing in Makefiles

In an ideal world, all programs would execute to completion successfully every time. Alas, this is not an ideal world. Despite our best efforts, the programs we develop do fail from time to time, for one reason or another. Thus, we use exit (or return) codes to determine whether a program failed and, often, what caused the failure.

In the OS/2 environment, the command processor supports the `errorlevel` statement. With this mechanism, we can test a program's exit code and take appropriate action based on the result. Consider the following simple example:

```
runxyz
if errorlevel 1 doprtscn
```

Here we execute the program `runxyz`, then test its exit code with the `errorlevel` statement. If `runxyz` returns an exit code of 1, we execute a program that writes the contents of the screen to the printer. If the exit code is any other value (or if `runxyz` does not return an exit code), no other action is performed.

This technique can be used effectively in batch, or command, processing. Unfortunately, it does not work for makefiles. At least, not in the form shown.

Let's say that we write a makefile in which we execute the program `runxyz`. The program fails. Now what?

Well, if we invoked NMAKE without the `i` option, NMAKE would stop when it received the exit code from `runxyz` indicating the program had failed. On the other hand, if we used the `i` option (or the `.IGNORE:` pseudotarget), NMAKE would ignore the error reported by `runxyz` and continue processing the rest of the makefile.

Neither action suits our purpose. How then can we execute a program from within a makefile, determine whether the program worked correctly, and have NMAKE take some alternative action based on the success or failure of the program?

The answer is to use a little-known feature of NMAKE on OS/2, which we will call *command level processing.* Command level processing allows us to spawn a process and to evaluate the results of the process, all at a command level. The following example represents a line of code from

a makefile showing how we can use command level processing in this environment:

```
(runxyz || if errorlevel 1 doprtscn)
```

When we have information that we want to examine — such as whether a program has completed successfully — we place that information within parentheses, thus spawning a child process in OS/2. This allows NMAKE to evaluate the return code from the program (for example, `runxyz`) directly.

Let's carry this example a step further. Suppose that `runxyz` can return either of two different error level codes, and we want to perform different operations depending on which error level the program returns. Before implementing this more complex solution, we must understand that error level values are cumulative in command processing. This means, for example, that if error level 2 is set, so is error level 1. With this in mind, here is the line from a makefile that allows us to direct NMAKE to take alternative actions based on the return code from `runxyz`:

```
(runxyz || if errorlevel 2 writeerr ||
           if errorlevel 1 doprtscn)
```

In this example, if `runxyz` returns error level 2, both `writeerr` and `doprtscn` are executed, in that order. (Note that we tested for error level 2 before error level 1.) If `runxyz` returns error level 1 only (that is, not error level 2), then only `doprtscn` is executed. If neither error level is set, no other action is performed.

By combining this feature of NMAKE with the concept of batch files, we can achieve the equivalent of OS/2 command level processing indirectly through our makefiles.

NMAKE and the tools.ini File

The latest version of the IBM C compiler provides for a special file, called `tools.ini`, which NMAKE can use. When we invoke NMAKE, before it begins to execute the makefile it first reads `tools.ini` (when the file is present) and uses the macros and inference rules it finds there. There is an exception to this general rule: if a makefile and `tools.ini` both contain a definition for a macro with the same name or an inference rule with the

same extensions, the definition in the makefile overrides the definition in `tools.ini`.

NMAKE looks for `tools.ini` in the current directory. If `tools.ini` is not in the current directory, we can use the `set` command to create an environment variable called `INIT`. (We learned about environment variables and the `set` command in Chapter 4.) We can define `INIT` to specify the path to the directory that contains initialization files, including `tools.ini`. Now NMAKE can find `tools.ini` by searching the directory that `INIT` specifies.

We can use `tools.ini` to customize default options for NMAKE. Suppose, for example, that we want to use a file with the extension `.inc` to create another file with the extension `.tex`. We can create an inference rule so NMAKE knows how to perform this task for us:

```
.inc.tex:
        include  <$*.inc  >$*.tex
```

If we need this rule in several different makefiles, we can omit the rule from the individual makefiles and add it to `tools.ini`:

```
[NMAKE]
.inc.tex:
        include  <$*.inc  >$*.tex
```

Thus, we can use `tools.ini` the same way we use the `include` directive, which we learned about earlier. Is one technique preferred over the other? In terms of results, no; they are the same. However, consider this: If we use the `include` preprocessor directive, we must not only create another file, we must also remember to put the `include` statement in the makefile. NMAKE, on the other hand, looks for and reads `tools.ini` automatically (unless we use the `r` option), requiring no extra effort on our part.

The online help information from IBM provides more specific details about the `tools.ini` file and how to use it.

Macro Precedence

A macro defined on the command line has the highest precedence in NMAKE. At the next level of precedence are macros defined in the makefile. A macro defined in the makefile has precedence over inherited macros. (Inherited macros are macros that are equivalent to every environment variable. NMAKE makes a copy of environment variables and gives them

uppercase names.) At the next level below inherited macros are macros defined in the `tools.ini` file, followed by predefined macros.

For example, if a macro name appears in `tools.ini`, then NMAKE will use the value for that macro from `tools.ini` as long as a macro with the same name does not appear either on the command line or in the makefile and is not an inherited macro.

Let's consider a specific case. `CFLAGS` is a predefined macro that we discussed in Chapter 3. NMAKE will use its predefined value only if `CFLAGS` does not appear on the command line, is not defined in the makefile, is not an inherited macro, and does not appear in `tools.ini`. This is because predefined macros have the lowest precedence in NMAKE.

Using NMAKE to Develop GUI Applications

More and more applications are being developed for graphical user interfaces (GUIs) such as Presentation Manager (PM) from IBM, which runs on OS/2 systems. As we demonstrate in this section, programmers who use PM libraries and resources to develop software can combine those tools with NMAKE to create GUI applications.

In comparison to the basic OS/2 program development we've discussed so far, PM-type development requires a different approach to creating the various objects and the final executable file. The IBM C Set/2++ Compiler for OS/2 provides many of the libraries that we need to create a 32-bit PM application.

If we wish to develop a PM application, we must have:

- PM compiled program(s)
- OS/2 libraries — `os2386.lib`, `os2.lib`
- Any resource files (`.res` files) that are a part of the executable
- Module definition file

You can find detailed information about these items in the IBM C Set/2++ Version 2.0 reference manuals. (You can also find all the information you need to create PM and other types of OS/2 applications in the IBM C Set/2++ libraries.) Here we want to highlight just a few important points:

- You must use the IBM C Set/2++ libraries for OS/2 when you link your application files. These libraries files can be given as input to the linker in any order.

- You must use the IBM Resource Compiler (RC) to define and compile any resource files that your application will use. The Resource Compiler from IBM comes with OS/2 Version 2.x as part of the base operating system. A resource file is appended to the end of the executable and is considered read-only by the program. (See Figure 6.1.) The resulting resource file can be created from multiple source files that represent different sections of the application.

- The linker uses the module definition file to determine what type of executable to create. The module definition file is the *only* way you can tell the linker that you want to create a PM application. Without a module definition file, the linker creates only text mode applications.

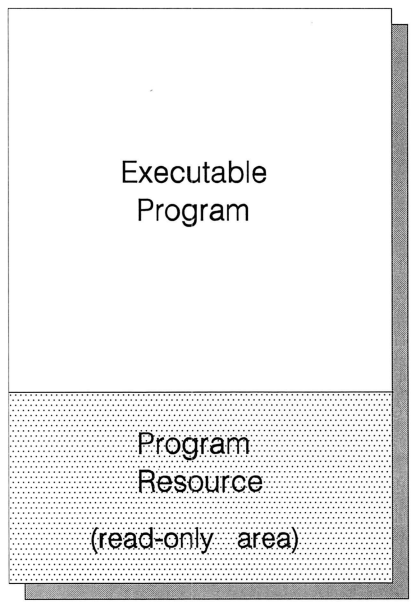


Figure 6.1. Presentation Manager GUI Format

Now that we have some understanding of the basic requirements, we are ready to look at an example. The following makefile (`pmsamp.mak`) creates

a window with text and two pull-down menus. One accesses the dialog feature, and the other exits the program. (This example demonstrates many of the *make* and NMAKE functions we've already discussed, as well as the development of a GUI application.)

```
#
# IBM Makefile for OS/2
# Presentation Manager Example
#
# Mastering Make: A Guide to Building Programs on
# DOS, OS/2, and UNIX* Systems
# Tondo, Nathanson, Yount
#
# Prentice Hall (C) 1994
#

#
# Define suffixes
#

.SUFFIXES:
.SUFFIXES: .c .obj .exe .rc .res .dlg .def

#
# Define macros
#

CC      = icc
LINK    = link386
LIBS    = os2386 os2
RC      = rc
TARGET  = pmsamp.exe
LIBPATH = h:\ibmcpp\lib
CINC    = -I.
RCFLAGS = -r

#
# Test for defined macro
#
```

```
!ifndef DEBUG
DEBUG=yes
!endif

#
# Define macros for compiling and linking
#

!if "$(DEBUG)" == "yes" || "$(DEBUG)" == "YES"
CFLAGS  = -c -Gd- -Se -Re -ss -Ms -Ti+ -W3 -G3
LFLAGS  = /co
!else
CFLAGS  = -c -Gd- -Se -Re -ss -Ms -W3 -G3
LFLAGS  =
!endif

#
# Define inference rules
#

.c.obj:
        $(CC) $(CFLAGS) $(CINC) $*.c

.rc.res:
        $(RC) $(RCFLAGS) $*.rc

#
# Define objects used to create the target
#

OBJS = $(TARGET:.exe=.obj)

#
# Define the target to build
#

$(TARGET): $(OBJS) $(TARGET:.exe=.def) \
           $(TARGET:.exe=.res)
```

```
        $(LINK) $(LFLAGS) @<<$(TARGET:.exe=.lnk)
$(OBJS)
$@
NUL
$(LIBS)
$(TARGET:.exe=.def)
<<keep
        $(RC) $(TARGET:.exe=.res) $@

#
# Define module definition file used to create executable
#

$(TARGET:.exe=.def):
        @echo $@ @<<$@
; Mastering Make - OS/2 PM Module Definition File Example

NAME         PMSAMP      WINDOWAPI

DESCRIPTION  'Mastering Make Example - IBM'

PROTMODE
<<keep

#
# Define indirect dependencies
#

$(TARGET:.exe=.obj): $(TARGET:.exe=.c)

$(TARGET:.exe=.res): $(TARGET:.exe=.rc) \
                     $(TARGET:.exe=.dlg)
```

This is certainly the most complex makefile we've seen so far. Let's take a closer look at its various parts.

The first section of our makefile defines information relating to the behavior of the makefile. The `.SUFFIXES:` pseudotarget tells the makefile

the types of files we will be using. This allows us to use the inference rules that are built into the NMAKE command as well as those we define later in the makefile.

In the next three sections we define the various macros we plan to use in our makefile, such as the libraries to be linked in and the name of the target file. Because we are interested in whether the makefile will build a debug or a production version of the program, we test to see if the macro `DEBUG` is defined. If `DEBUG` is not defined, then the definition of the macro becomes `yes`. We will use this macro to determine whether we are going to build a debug or production version of the program. The result of the test allows us to set the correct values for the other compile- and link-related macros.

Next, we want to make sure that NMAKE knows how to create object (`.obj`) and resource (`.res`) files, so we define inference rules for them using some of the suffixes we defined at the beginning of the makefile. Finally, after defining the macro for the object files, we reach the target that we want to build. Notice that we include as dependencies not only the object file `pmsamp.obj` (macro `OBJS`), but also the resource file `pmsamp.res` (macro `$(TARGET:.exe=.res)`) and the module definition file `pmsamp.def` (macro `$(TARGET:.exe=.def)`).

Now we come to the heart of this makefile. You may wonder why we have included the module definition file (`.def`) in the dependency list for an executable. Before we address that question, notice first that after defining the target, we use NMAKE's in-line response feature to perform two separate and distinct functions. In the first instance, we use in-line response to provide information to the linker. (We saw this usage earlier in this chapter.) But the second instance — ah, this is what catches our attention. Here we use in-line response to define a pseudotarget that effectively causes NMAKE to create the module definition file for us! This strategy assures us that two conditions — critical to our PM application — are met:

- The module definition file is present.
- The module definition file is current.

Somewhat as an anticlimax, we conclude the makefile by defining the indirect dependencies for the object file `pmsamp.obj` and the resource file `pmsamp.res`.

Notice that the `$(TARGET:.exe=.dlg)` file is a part of the direct dependency for `$(TARGET:.exe=.res)`. This is because when the resource file is compiled, it includes all dialog files (`.dlg`) as part of the source file.

Refer to the Presentation Manager sample source file (`.rc`) to see how the include process occurs. We must define the dialog file as we would any include file for a source file.

As we see from this example, no mystery attaches to using NMAKE to create a GUI application. The makefiles we write to build Presentation Manager applications are very much like any other makefile. We use the same fundamentals and similar techniques. As in any development project, we need only take care that our program conform to the requirements of the environment.

The Presentation Manager example we have just discussed presents a single point of view of how best to use NMAKE in a PM development effort. Refer to IBM's documentation for IBM C Set/2++ for more information about PM development and C++ applications. The complete source code for this example appears in Appendix G.

Borland's MAKE on OS/2

We begin this section by showing how to invoke MAKE. Then we explore how some of the topics we met earlier — such as options, dependencies, continuation lines, and environment variables — apply specifically to Borland's MAKE. New topics introduced in this section include preprocessor directives, input files and in-line response files, and the `builtins.mak` file. We conclude with some tips for using Borland's MAKE to develop applications for a graphical user interface (GUI) environment.

Borland's C++ for OS/2 Version 1.0 provides an interactive development environment (IDE), which is a 32-bit Presentation Manager utility. If we select "Create a makefile" from within IDE, the utility creates a stand-alone makefile that can be used outside IDE. Borland's *IDE User's Guide* explains how to use the utility.

Invoking MAKE

Here is the format for invoking MAKE on the command line:

```
make  [options]  [targets]
```

The brackets (`[ ]`) around a field name indicate that the field is optional. Thus, the two fields are optional for MAKE.

Command Line Options

MAKE supports an extensive list of command line options. If you are familiar with other versions of *make*, some of these options may look familiar. However, a number of MAKE options differ from those supported in other versions of *make*.

MAKE is case sensitive with respect to options; that is, MAKE treats `s` and `S` as two different options. Remember that when we specify an option on the command line we must precede the option letter with a dash (`-`) or a forward slash (`/`). Notice that when an option takes an argument we do not put any space between the option and the argument. (One exception is `f`; see below.)

Here are the options that MAKE supports:

`?`	Display help information
`a`	Perform automatic dependency check on all object files
`B`	Build all targets regardless of time stamp (file dates)
`d`*directory*	Identify where to write the swap file; *directory* can include a drive letter
`D`*identifier*	Define the value of *identifier* to be 1
`D`*identifier*`=`*string*	Define the value of *identifier* to be *string*
`e`	Let environment variables override macro definitions
`f`*filename*	Specify the makefile to be *filename*; space between `f` and the filename is optional (`f` *filename*)
`h`	Display help information
`i`	Ignore exit/error codes
`I`*directory*	Search for include files in *directory*
`K`	Retain any temporary file(s) created by MAKE

`m`	Display time and date stamp for each file as the file is being processed
`n`	Display commands but do not execute them
`N`	Change the behavior of in-line response processing to be compatible with NMAKE from Microsoft
`p`	Display all macros and inference rules before processing the makefile
`r`	Ignore rules that are defined in the `builtins.mak` file
`s`	Suppress command display while executing
`S`	Swap MAKE out of memory while executing commands
`U`*identifier*	Undefine *identifier* if previously defined
`W`	Write the current specified nonstring options to `make.exe`

Preprocessing Directives

A preprocessing directive is a command or an instruction that allows us to conditionally execute other commands or to conditionally select parts of a makefile. When MAKE reads a makefile, it processes any preprocessing directives it finds before it executes commands associated with targets.

A preprocessing directive begins with an exclamation mark (`!`) and must appear at the beginning of the line. That is, we must place the `!` in the first column on the line. However, after the `!` we can use blank spaces and/or tabs, if we wish, to enhance readability of the file.

Here are the preprocessing directives that MAKE supports:

`error` *message*	Display error message
`include <`*filename*`>`	Insert another file as part of this makefile; MAKE searches for the file in the directories specified in the environment variable `INCLUDE`
`include "`*filename*`"`	Insert another file as part of this makefile

`if` *expr*	Execute statements following `if` when *expr* evaluates to nonzero
`elif` *expr*	Execute statements following `elif` when the preceding `if` evaluates to zero and this *expr* evaluates to nonzero
`else`	Execute statements following `else` when *expr* in the preceding `if` (and in the preceding `elif`, when present) evaluates to zero
`endif`	Mark the end of an `if` directive
`undef` *name*	Undefine *name* (same as the `U`*identifier* option on the command line)

We can use conditional operators with the `if`, `elif`, and `else` preprocessing directives. The conditional operators that MAKE supports are the same as those supported in the C programming language. They are:

- Unary operators
  - `-` arithmetic negation
  - `~` bitwise complement
  - `!` logical NOT
- Binary operators

`+`	addition	`>>`	right shift
`-`	subtraction	`<<`	left shift
`*`	multiplication	`&`	bitwise AND
`/`	division	`|`	bitwise OR
`%`	modulus	`^`	bitwise XOR
`&&`	logical AND	`>=`	greater than or equal
`||`	logical OR	`<=`	less than or equal
`>`	greater than	`==`	equality
`<`	less than	`!=`	inequality

- Ternary operator
  - `? :` conditional expression

We can use four of the binary operators listed above to compare string expressions as well as numeric expressions. The operators that we can use for string comparison are `>=`, `<=`, `==`, and `!=`.

To see how we can apply preprocessing directives, let's start with a variation of a makefile from Chapter 3. We've adapted the makefile to work with Borland's MAKE. Here is the modified makefile:

```
CFLAGS          =  -c
OBJS            =  prog1.obj  prog2.obj
TARGET          =  prog.exe

$(TARGET).exe:  $(OBJS)
        $(CC)  -e$@  $(OBJS)
```

Now let's modify this makefile to use directives. Here is what we want to achieve:

- Include another makefile (called `sample.mak`) as part of our makefile. The macros we want to use will be defined in this file.
- Use the `$d` predefined macro (which tests for a defined macro) to determine if some action should be performed (see next item).
- If the `DEBUG` macro has been defined, add the `-O-` and `-v` compiler options to `CFLAGS` (`-O-` indicates that we want to disable optimization; `-v` turns source debugging on).

Here is the makefile in its new form:

```
# Define suffixes

.suffixes:
.suffixes:  .c  .obj  .exe

# Include the file sample.mak

!include  "sample.mak"

# Test for defined macro

!if $d(DEBUG)
CFLAGS          =  $(CFLAGS)  -O-  -r-
!endif
```

```
# Target and dependencies

$(EXEC_NAME).exe:  $(OBJS)
        $(CC)  -e$@  $(OBJS)
```

Let's look more closely at the new features of this makefile, beginning with the `include` directive. We have used the `include` directive

```
!include  "sample.mak"
```

to tell MAKE to read the file `sample.mak` into the current makefile. When MAKE begins to process the makefile, it appears as if the contents of `sample.mak` were always present. Effectively, they are: the directive brings them in. Here are the contents of `sample.mak`:

```
# Define macros

CFLAGS          =  -c -ms
OBJS            =  prog1.obj  prog2.obj
EXEC_NAME       =  prog
```

By using the `include` directive, we can easily create part of a makefile — a sequence of statements — that is common to various makefiles we use. Then, whenever we need this particular sequence of statements, we tell MAKE to include the file that contains them.

Let's look now at how we've used the `if` directive. Remember that this directive tells MAKE to evaluate an expression; that is, to determine whether the expression is true (the return value is nonzero) or false (the return value is zero). For our immediate interest, the directive statement

```
!if $d(DEBUG)
```

asks MAKE whether a macro named `DEBUG` has been defined. (Notice that the predefined macro `$d` uses a `d` — not a `D` — and should not be confused with the `D`*identifier* option from the command line. Remember that MAKE is case sensitive.)

We can define the macro `DEBUG` in the makefile:

```
DEBUG  =  3
```

Or we can define the macro on the command line using the `D`*identifier* option:

```
make  -DDEBUG=3
```

If DEBUG is defined, MAKE executes the statements between the if and the endif directives. In our case, MAKE would change CFLAGS to be defined as:

```
CFLAGS  =  -c  -3  -O  -v
```

It is important to notice that CFLAGS is left justified on the line; that is, the C of CFLAGS appears below the !. We must begin CFLAGS in the first column of the line because it is a macro name, not a command. Remember that only commands are indented on the line (with one or more spaces or tabs).

If DEBUG is not defined, then MAKE does not execute the statements between the if and the endif directives; CFLAGS remains as originally defined.

Let's consider using the DEBUG macro in a different way. With the if and the endif directives, we can explicitly test whether DEBUG has been defined to be 3 (or any other value we want to test for):

```
!if $(DEBUG) == 3

      ⋮

!endif
```

We can use an else directive following the if directive:

```
!if $(DEBUG) == 3

      ⋮

!else

      ⋮

!endif
```

This variation means that when DEBUG is defined to be 3, MAKE executes the statements between the if and the else directives. When DEBUG is defined to be some value other than 3 (or is not defined at all), MAKE executes the statements between the else and the endif directives.

Let's add our debug flags to CFLAGS in the first case; let's keep CFLAGS simple in the second case:

```
!if $(DEBUG) == 3
CFLAGS  =  -c -3 -O -v
!else
CFLAGS  =  -c -3
!endif
```

Notice in these examples of the `if` directive that we use the notation `$( )` around `DEBUG`. This is because we are interested in the value of the macro `DEBUG`, and the notation that tells MAKE to use the value of a macro is `$( )`.

Now let's consider a variation on our earlier directive statement:

```
!if !$d(DEBUG)
```

With this `if` directive using the `$d` predefined macro, we are asking MAKE to evaluate whether the `DEBUG` macro is defined. More specifically, by using the syntax shown, we are checking for the nonexistence of the macro (the second `!` in the statement is a NOT operator). In other words, if `DEBUG` is not defined, MAKE will return a true (nonzero) value to the preprocessing directive.

As with any `if` statement, we can use the results of this evaluation to conditionally perform some specific task. In the following example, the action MAKE takes is based on whether the macro `DEBUG` has or has not been defined:

```
!if !$d(DEBUG)
DEBUG  =  3
!endif
```

If the macro already is defined, the expression evaluates to false, and MAKE ignores the new definition given between the `if` and `endif` directives. However, if the macro is not defined, the condition given to the `if` directive evaluates to true, and MAKE defines the macro with the value `3`.

We can write more than one statement following an `if` or an `else` directive. For example,

```
!if $d(DEBUG)
CFLAGS  =  -c  -3  -O-  -v
MESSAGE = "DEBUG has been defined"
!       if !$d(LISTING)
CFLAGS  =  $(CFLAGS)
```

```
!       else
CFLAGS  =  -y  $(CFLAGS)
!       endif
!else
CFLAGS  =  -c  -3
MESSAGE = "DEBUG has NOT been defined"
!       if !$d(LISTING)
CFLAGS  =  $(CFLAGS)
!       else
CFLAGS  =  -y  $(CFLAGS)
!       endif
!endif
```

In this example we are testing first whether the DEBUG macro has been defined. If DEBUG has been defined, and we assign one set of compiler options to CFLAGS and we assign a string to MESSAGE; otherwise, we define CFLAGS for a different set of options, and we assign a different string to MESSAGE. In both cases, we test whether the LISTING macro has been defined.

Notice we have an `if` within an `if` (nested `if` statements) when we test whether the LISTING macro has been defined. In this case, if LISTING has not been defined, we keep the current settings; otherwise, we add the compiler option y to produce line number information.

This example highlights two additional points of interest. First, each `if` directive must conclude with its own corresponding `endif` directive. Second, as long as we place the `!` in the first column of the line, we can use spaces or tabs to enhance readability.

Now let's make some more changes to the makefile to further illustrate the use of directives. This time our criteria are:

- Look in the current directory for the source files.
- Put the object files in a subdirectory called `sample2` (we will use the o compiler option to accomplish this).
- If the DEBUG macro is defined, change the compiler options to include debug information. Otherwise, use the value of CFLAGS as defined in the include file `sample.mak`.

To meet these criteria, we change our makefile so that it looks like this:

```
# Define suffixes

.suffixes:
.suffixes:  .c  .obj  .exe

# Define macros

.path.c   =  .
.path.obj =  .\sample2

OBJS = $(.path.obj)\prog1.obj $(.path.obj)\prog2.obj

# Include the file sample.mak

!include     "sample.mak"

# Test for defined macro

!if $d(DEBUG)
CFLAGS    =  $(CFLAGS)  -O-  -v
!endif

# Define inference rules

.c.obj:
        $(CC) $(CFLAGS) -o$(.path.obj)\$*.obj $*.c

# Target and dependencies

$(EXEC_NAME).exe:  $(OBJS)
        $(CC)  -e$@  $(OBJS)
```

Now let's look at the changes we've made. First, we added two `.path` statements. These statements tell MAKE the paths to use when searching for files with the `.c` extension and the `.obj` extension; specifically, the `.c` files for this makefile are in the current directory, and the `.obj` files are in a subdirectory called `sample2`.

Next, we changed the `OBJS` macro. We redefined `OBJS` to tell MAKE that the path for the object files is not the current directory, but is the

directory defined for `.path.obj`.

Finally, we changed the compiler command line associated with the inference rule for building `.obj` files from `.c` files. Now when MAKE compiles a `.c` file, it places the output (that is, the `.obj` file) in the directory defined for `.path.obj`.

From this brief discussion of preprocessing directives, we've gained some insight into their usefulness in makefiles. Different circumstances will suggest other uses. For more information about this versatile feature, refer to the documentation that Borland provides for MAKE.

Special Cases

Unlike other versions of the *make* utility, Borland's MAKE does not support escape sequences. This fact sets up an interesting situation.

We already know that a backslash (\) at the end of a line tells MAKE that the next line in the makefile is a continuation of the line on which the \ occurs. Suppose, however, that we want to copy a file created in our makefile to the root directory of the system, which also is represented by the character \.

How does MAKE know the difference between the case where \ indicates a continuation line

```
prog1.obj: prog1.c    \
        prog1.h
```

and the case where it does not:

```
copy  $<  \
```

The answer, fortunately, is simple: we can use a double backslash (\\) to tell MAKE that it should see a literal \ and not a line continuation indicator. Thus, when MAKE sees

```
copy  $<  \\
```

it knows to copy the file represented by `$<` to the root directory.

Creating Dependencies Dynamically

In Chapter 2 we mentioned the need for performing dependency checking. We are ready now to explore that concept further.

We well know by now that an important use of MAKE is to maintain program modules within a development project. We also recall (from Chapter 2) that a source file may include other files.

We sometimes call these other files *header files* because they contain information that typically appears at the beginning, or head, of a program file. We also refer to these files as *include files*, for the obvious reason. In C language programs, the header (or include) files usually are recognizable by the `.h` extension to the file name; in assembler, the `.inc` extension is used. An include file may contain `#define` statements (C language) or `equate` statements (assembly language) that convey information to the file in which it is included.

Depending on the project, it may be more the rule than the exception that program modules use include files. To illustrate how this affects dependencies in a makefile, let's look again at the Borland version of the makefile we used at the beginning of this section:

```
# Define suffixes

.suffixes:
.suffixes:  .c  .obj  .exe

# Define macros

CFLAGS          =  -c  -3
OBJS            =  prog1.obj  prog2.obj
TARGET          =  prog.exe

# Target and dependencies

$(TARGET).exe:  $(OBJS)
        $(CC)  -e$@  $(OBJS)

prog1.obj:      prog1.c

prog2.obj:      prog2.c
```

(The lines with `prog1.c` and `prog2.c` ordinarily would not appear in the makefile because of `.suffixes:` and the inference rule; we are showing them in this case to illustrate the need for dependency checking.)

Based on this makefile, the only dependency for `prog1.obj` is `prog1.c`. But suppose that `prog1.c` uses the include file `prog1.h`. The relevant line in the makefile

```
prog1.obj:  prog1.c
```

gives no indication that `prog1.h` has anything to do with this dependency.

Now let's say that we have changed `prog1.h` — but not `prog1.c` — since the last time our makefile produced the `.obj` file for `prog1.c`. Because `prog1.c` includes `prog1.h`, we want MAKE to recompile `prog1.c` when either it or `prog1.h` is modified. But MAKE does not know this. In this situation, MAKE does not know to check the date and time of `prog1.h`. It checks the date and time of `prog1.obj` and of `prog1.c` only and does not recompile `prog1.c` until `prog1.c` is modified.

An obvious solution to the apparent dilemma presents itself. We need simply to change our makefile to show that `prog1.obj` depends on both files:

```
prog1.obj:  prog1.c  prog1.h
```

It is clear now that `prog1.h` is a direct dependency for `prog1.obj`.

It is vital, as this example shows, that we state all the dependencies for every file we need to work with in the makefile.

The importance of this requirement cannot be overstated; however, it is not always easy to remember every include file in every program module. But because we are using the Borland version of the *make* utility, we are in luck!

One feature Borland provides is the availability of built-in dependency checking for all source files. Recall from page 160 that the `a` option instructs MAKE to perform dependency checking for us. When we use this option, the source programs are checked for include files and an internal dependency list is created for MAKE to use. The makefile itself does not change, unless we use the `K` option along with the `a` option. (Remember that the `K` option tells MAKE to keep the temporary files it creates; otherwise, they are automatically removed.)

Thus, to build a complete and accurate dependency list for our makefile, we specify the `a` option when we invoke MAKE:

```
make  -a
```

Because we issued the `make` command with the `a` option, MAKE determines whether any source files contain include files (and, if so, searches the current directory and the directories listed in the `INCLUDE` environment variable), then builds the `.obj` targets accordingly.

Borland provides an alternative to using the `a` option on the command line. Instead we can insert in the makefile the statement:

```
.autodepend
```

This statement provides the same functionality as when we invoke MAKE with the `a` option as shown previously. This technique offers the advantage of ensuring that built-in dependency checking is performed every time the makefile is executed. In contrast, if we rely on invoking the feature from the command line, we may inadvertently omit the `a` option when we issue the `make` command.

In either case, we see that the job of building the dependency list for any makefile is much easier to accomplish when we use the built-in dependency checking feature. This feature is so helpful, in fact, that we are well-advised to use it liberally, either by means of the `a` option on the command line or the `.autodepend` statement within the makefile.

If we want to generate the list of dependencies within the makefile, we can use the `depends` utility described in Appendix I. For an example, refer to page 145.

Input Files and In-Line Response Files

Remember the very first makefile for MAKE we looked at in Chapter 1? Here it is again, modified to run on OS/2:

```
prog.exe:  prog1.obj  prog2.obj
        bcc  -eprog.exe  prog1.obj  prog2.obj

prog1.obj:  prog1.c
        bcc  -c  -3  prog1.c

prog2.obj:  prog2.c
        bcc  -c  -3  prog2.c
```

Notice that we linked the `.obj` files and produced the target `prog.exe` using the command line:

```
bcc  -eprog.exe  prog1.obj  prog2.obj
```

An alternative to using `bcc` to link is using the `tlink` command explicitly:

```
tlink  prog1.obj  prog2.obj, prog.exe ;
```

Notice the semicolon (`;`) at the end of the `tlink` command line. The purpose of the semicolon is to instruct the linker to use default values for the link options instead of prompting the user for them. If we want to use only some (or none) of the default values, we simply include on the command line the options we want before (or instead of) the semicolon.

The following command line shows a case in which we have decided not to use all the default options:

```
tlink  prog1.obj  prog2.obj,  prog.exe,  ,  c2 ;
```

(The `tlink` command accepts the names of the `.obj` files separated by either blanks or plus signs (`+`).)

This `tlink` command line includes an option that was not present in the earlier case: `c2`, which tells the linker which library to bring in to link with our object files.

Notice that we retained the semicolon at the end of the command line. Without it, the linker might prompt for other options that we haven't specified. However, because the semicolon is present, the linker knows to use default values for the rest of the link options.

Our example `tlink` command line is pretty compact, but command lines can easily become quite lengthy. Command lines in makefiles can be particularly bothersome because *make* requires a backslash (\) at the end of a line to indicate that the next line is a continuation of the current line.

The solution to dealing with lengthy, unwieldy command lines in a makefile is simple. Using our example, we place the parameters to the `tlink` command in a file. Then, in the makefile, we tell the linker to use the file instead of depending on a list of parameters on the command line.

To get an idea of how this works, let's put our link parameters into a file called `makefile.lnk`. Here they are again:

```
prog1.obj  prog2.obj,  prog.exe,  ,  c2 ;
```

We've just created an input file to the linker. (An input file is also referred to as a *response file* or an *autoresponse file.*) To have the linker read the input file, we precede the name of the input file with the "at" character (`@`). (Note that this use of `@` is distinct from the predefined macro `$@`.) This character tells the linker to use the link information stored in the file and not to prompt for input. Now the `tlink` command line within the makefile looks like this:

```
tlink  @makefile.lnk
```

It's easy to see, even with our compact example, how helpful an input file can be. Its usefulness is even more evident when a large number of files and linker options must be used.

Now that we understand the basic concept of the input file to a linker, let's consider how we can make it even more useful. For example, what if we want to use a macro in an input file? We can do it: the linker is aware of environment variables.

Recall that in Chapter 4 we created an environment variable called `ROOTDIR` and showed how we could use it as if it were a macro. Here's an example of what our input file would look like using the environment variable as a macro:

```
prog1.obj  prog2.obj,  $(ROOTDIR)\prog.exe,  ,  c2 ;
```

Closely related to input files are in-line response files. You can see the similarity in the following example:

```
tlink  @&&|
prog1.obj  prog2.obj,  $(ROOTDIR)\prog.exe,  ,  c2 ;
|
```

In the earlier example, we told the linker (by means of the `@` character) to get its parameters from a file named `makefile.lnk`. In the example just above, we've inserted the notation `&&|` between the `@` character and the input to `tlink`. This means that `tlink` will get its input from a temporary

file (indicated by `&&`) that contains the items beginning after the vertical line character (`|`) and continuing up to the next occurrence of the character. A file created in this manner is called an *in-line response file.*

Note that the in-line response delimiter can be any character that is not special to *make.* In our example we used the `|` character; however, we could have selected any character for our delimiter.

We can construct an in-line response with as many lines as we like. To do this, we simply add lines — without backslashes — and use the second occurrence of our delimiter (the `|` character in the example above) to indicate that we have completed the in-line response:

```
tlink  @&&|
prog1.obj  prog2.obj
$(ROOTDIR)\prog.exe
nul
c2 ;
|
```

In each of these examples, MAKE generates a name for the temporary file it creates. The name of these files typically takes the form `MAKExxxx.$$$` (where `xxxx` is a four-digit number that MAKE generates). If we wish, we can use the `copy` command along with the in-line response function to create a file with a name that we choose. Here is how we can assign a specific name to this type of file:

```
copy &&|
prog1.obj  prog2.obj,
$(ROOTDIR)\prog.exe
nul
c2 ;
| makefile.lnk
```

In this example, the command copies the items between the delimiters (in this case, between `|` and `|`) to a file named `makefile.lnk`. We can use this file in the link statement as we did earlier:

```
tlink  @makefile.lnk
```

Note that a file created by in-line response is a temporary file. After executing the makefile, MAKE removes any temporary files that were

created in the process. This is true whether we assign a name or accept the generated file name.

But what if we want to save the in-line response file for some future use? If we want to keep an in-line response file, we must invoke MAKE with the `K` option. This option prevents all temporary files that are created during execution of the makefile from being deleted when the makefile completes.

You can find out more about using input files and the in-line response function from the documentation provided with your compiler. Refer to the *Borland C++ Tools and Utilities Guide.*

Using Command Level Processing in Makefiles

In an ideal world, all programs would execute to completion successfully every time. Alas, this is not an ideal world. Despite our best efforts, the programs we develop do fail from time to time, for one reason or another. Thus, we use exit (or return) codes to determine whether a program failed and, often, what caused the failure.

In the OS/2 environment, the command processor supports the `errorlevel` statement. With this mechanism, we can test a program's exit code and take appropriate action based on the result. Consider the following simple example:

```
runxyz
if errorlevel 1 doprtscn
```

Here we execute the program `runxyz`, then test its exit code with the `errorlevel` statement. If `runxyz` returns an exit code of 1, we execute a program that writes the contents of the screen to the printer. If the exit code is any other value (or if `runxyz` does not return an exit code), no other action is performed.

This technique can be used effectively in batch, or command, processing. Unfortunately, it does not work for makefiles. At least, not in the form shown. But there is a way around this difficulty.

Let's say that we write a makefile in which we execute the program `runxyz`. The program fails. Now what?

Well, if we invoked MAKE without the `i` option, MAKE would stop when it received the exit code from `runxyz`, indicating the program had failed. On the other hand, if we used the `i` option (or the `.ignore:` pseudotarget),

MAKE would ignore the error reported by `runxyz` and continue processing the rest of the makefile.

Neither action suits our purpose. How then can we execute a program from within a makefile, determine whether the program worked correctly, and have MAKE take some alternative action based on the success or failure of the program?

The answer is to use a little-known feature of MAKE on OS/2, which we will call *command level processing.* Command level processing allows us to spawn a process and to evaluate the results of the process, all at a command level. The following example represents a line of code from a makefile showing how we can use command level processing in this environment:

```
(runxyz || if errorlevel 1 doprtscn)
```

When we have information that we want to examine — such as whether a program has completed successfully — we place that information within parentheses, thus spawning a child process in OS/2. This allows MAKE to evaluate the return code from the program (for example, `runxyz`) directly.

Let's carry this example a step further. Suppose that `runxyz` can return either of two different error level codes, and we want to perform different operations depending on which error level the program returns. Before implementing this more complex solution, we must understand that error level values are cumulative in command processing. This means, for example, that if error level 2 is set, so is error level 1. With this in mind, here is the line from a makefile that allows us to direct MAKE to take alternative actions based on the return code from `runxyz`:

```
(runxyz || if errorlevel 2 writeerr ||
           if errorlevel 1 doprtscn)
```

In this example, if `runxyz` returns error level 2, both `writeerr` and `doprtscn` are executed, in that order. (Note that we tested for error level 2 before error level 1.) If `runxyz` returns error level 1 only (that is, not error level 2), then only `doprtscn` is executed. If neither error level is set, no other action is performed.

By combining this feature of MAKE with the concept of batch files, we can achieve the equivalent of OS/2 command level processing indirectly through our makefiles.

MAKE and the builtins.mak File

MAKE supports the use of a general file called `builtins.mak`. When we issue the `make` command on OS/2, it automatically searches for this special file and, if it is present, includes the file as part of the makefile currently being executed. However, MAKE does not require `builtins.mak` to exist.

The `builtins.mak` feature provides a tool we can use to supplement the limited set of predefined macros that MAKE supports. We can also use this file to develop and maintain a set of inference rules (remember that MAKE has a limited set of predefined inference rules).

To make use of this feature, we create the file `builtins.mak`, which should reside in the same directory as `make.exe`, and include in it the macros and rules we want to use over and over again. We can illustrate this idea by making a few changes to our previous makefile (see page 167). We will remove the `include` directive and the inference rule and put them into `builtins.mak`. (Remember that once we remove the `include` directive, MAKE does not know about the `sample.mak` file or its contents.)

Here is our new makefile:

```
# Define suffixes

.suffixes:
.suffixes:  .c  .obj  .exe

# Define macros

.path.c   =  .
.path.obj =  .\sample2

OBJS = $(.path.obj)\prog1.obj $(.path.obj)\prog2.obj

# Test for defined macro

!if $d(DEBUG)
CFLAGS  =  $(CFLAGS)  -O-  -v
!endif

# Target and dependencies
```

```
$(EXEC_NAME).exe:  $(OBJS)
        $(CC)  -e$@  $(OBJS)
```

We see that this makefile still relies on the macros `EXEC_NAME` and `CFLAGS`; however, they are not defined in the makefile (we are assuming that the `DEBUG` macro was defined on the command line). Furthermore, we want MAKE to build `.obj` files from `.c` files by using a specific path, but we have not provided any rules for doing so. Nor have we told MAKE to include the contents of any other file that might contain this information.

How, then, does MAKE know what to do? Simply by reading the `builtins.mak` file, which we wrote to include the following lines:

```
CFLAGS      =  -c  -3
EXEC_NAME   =  prog

.c.obj:
        $(CC) $(CFLAGS) -o$(.PATH.obj)\$*.obj $*.c
```

The technique of using `builtins.mak` works exactly like the `include` directive with one significant difference: the lines in `builtins.mak` will be used in every makefile executed by MAKE on our system (unless we use the `r` option), instead of only in those makefiles in which we remember to use the `include` directive.

Macro Precedence

The order of precedence for macros in Borland's MAKE is simple: the last definition of the macro that MAKE sees is the definition that takes effect. Here are some specific instances that show how this works:

If a macro has been defined in `builtins.mak`, then MAKE uses that definition. If the same macro is redefined in the makefile, the latest definition takes effect.

A macro can be defined on the command line using the `D`*identifier* option. If we redefine the same macro in the makefile, that is considered its latest definition; again, the latest definition takes effect.

If a macro is defined on the command line and is redefined in `builtins.mak`, the redefinition from `builtins.mak` takes effect.

Using MAKE to Develop GUI Applications

More and more applications are being developed for graphical user interfaces (GUIs) such as Presentation Manager (PM) from IBM, which runs on OS/2 systems. As we demonstrate in this section, programmers who use PM libraries and resources to develop software can combine those tools with MAKE to create GUI applications.

In comparison to the basic OS/2 program development we've discussed so far, PM-type development requires a different approach to creating the various objects and the final executable file. The Borland C++ Compiler for OS/2 provides many of the libraries that we need to create a 32-bit PM application.

If we wish to develop a PM application, we must have:

- PM compiled program(s)
- Borland C++ OS/2 interface program — `c02.obj`
- OS/2 libraries – `c2.lib`, `os2.lib`
- Any resource files (`.res` files) that are a part of the executable
- Module definition file

You can find detailed information about PM and other types of OS/2 applications in the Borland C++ for OS/2 reference manuals. Here we want to highlight just a few important points:

- The Borland C++ OS/2 interface program is an object file that you must link with your application for it to be OS/2-compatible. This object file *must* be the first file given as input to the `tlink` command.
- You must use the Borland C++ libraries for OS/2 when you link your application files. The library files can be given as input to the `tlink` command in any order; however, they must follow the Borland C++ interface program described previously.
- You must use the Borland Resource Compiler (BRCC) and the IBM Resource Compiler (RC) to define and compile any resource files that your application will use. The Resource Compiler from IBM comes with OS/2 Version 2.x as part of the base operating system. A resource file is appended to the end of the executable and is considered read-only by the program. (See Figure 6.2.) The resulting resource file can

be created from multiple source files that represent different sections of the application.

- The linker uses the module definition file to determine what type of executable to create. The module definition file is the *only* way you can tell the linker that you want to create a PM application. Without a module definition file, the linker creates only text mode applications.

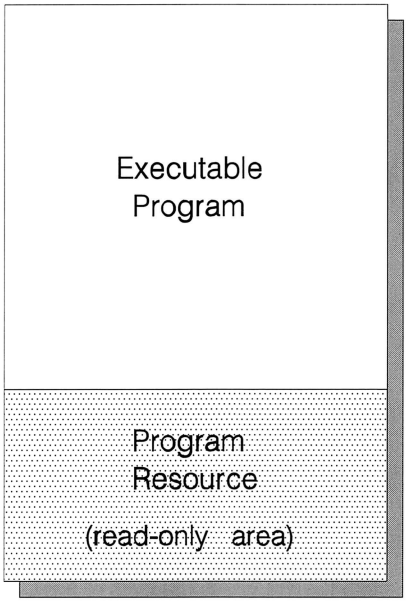


Figure 6.2. OS/2 GUI Format

Now that we have some understanding of the basic requirements, we are ready to look at an example. The following makefile (`pmsamp.mak`) creates a window with text and two pull-down menus. One accesses the dialog feature, and the other exits the program. (This example demonstrates many of the *make* and MAKE functions we've already discussed, as well as the development of a GUI application.)

```
#
# Borland Makefile for OS/2
# Presentation Manager Example
#
# Mastering Make: A Guide to Building Programs on
# DOS, OS/2, and UNIX* Systems
# Tondo, Nathanson, Yount
#
# Prentice Hall (C) 1994
#

#
# Define suffixes
#

.suffixes:
.suffixes: .c .obj .exe .rc .res .dlg .def

#
# Define other makefile directives
#

.autodepend

.path.obj = .

#
# Define macros
#

CC      = bcc
CINC    = -I.
LINK    = tlink
LIBS    = c2 os2
BRCC    = brcc
RC      = rc
RCFLAGS = -r
TARGET  = pmsamp.exe
```

```
#
# Test for defined macro
#

!if !$d(DEBUG)
DEBUG=yes
!endif

#
# Define macros for compiling and linking
#

!if $(DEBUG) == "yes" || $(DEBUG) == "YES"
CFLAGS  = -c -3 -v
LFLAGS  = -v -Toe -aa
!else
CFLAGS  = -c -3 -O1
LFLAGS  = -Toe -aa
!endif

#
# Define inference rules
#

.c.obj:
   $(CC) $(CFLAGS) $(CINC) $*.c

.rc.res:
   $(BRCC) $(RCFLAGS) $*.rc

#
# Define objects used to create the target
#

OBJS  = $(TARGET:.exe=.obj)

BCOBJ = c02.obj

#
```

```
# Define the target to build
#

$(TARGET): $(OBJS) $(TARGET:.exe=.def) \
           $(TARGET:.exe=.res)
        $(LINK) $(LFLAGS) @&&|
$(BCOBJ)  $(OBJS)
$@
nul
$(LIBS)
$(TARGET:.exe=.def)
|
        $(RC) $(TARGET:.exe=.res) $@

#
# Define module definition file used to create executable
#

$(TARGET:.exe=.def):
        @copy &&|
; Mastering Make - OS/2 PM Module Definition File Example

NAME         PMSAMP      WINDOWAPI

DESCRIPTION  'Mastering Make Example - Borland'

PROTMODE
| $@

#
# Define indirect dependencies
#

$(TARGET:.exe=.obj): $(TARGET:.exe=.c)

$(TARGET:.exe=.res): $(TARGET:.exe=.rc) \
                     $(TARGET:.exe=.dlg)
```

This is certainly the most complex makefile we've seen so far. Let's take a closer look at its various parts.

We begin by clearing MAKE's built-in list of suffixes, then use the `.suffixes:` pseudotarget to tell MAKE what suffixes we want to use. Next we define directives that tell MAKE to check the dependency of all files named in the makefile and where the object files are to be created.

In the next section we define the various macros we plan to use in our makefile, such as the libraries to be linked in and the name of the target file. Because we are interested in whether the makefile will build a debug or production version of the program, we test to see if the macro `DEBUG` is defined. If `DEBUG` is not defined, then the definition of the macro becomes `yes`. We will use this macro to determine whether we are going to build a debug or production version of the program. The result of the test allows us to set the correct values for the other compile- and link-related macros.

Next, we want to make sure that MAKE knows how to create object (`.obj`) and resource (`.res`) files, so we define inference rules for them using some of the suffixes we defined at the beginning of the makefile. Finally, after defining the macro for the object files, we reach the target that we want to build. Notice that we include as dependencies not only the object file `pmsamp.obj` (macro `OBJS`), but also the resource file `pmsamp.res` (macro `$(TARGET:.exe=.res)`) and the module definition file `pmsamp.def` (macro `$(TARGET:.exe=.def)`).

Now we come to the heart of this makefile. You may wonder why we have included the module definition file (`.def`) in the dependency list for an executable. Before we address that question, notice first that after defining the target, we use MAKE's in-line response feature to perform two separate and distinct functions. In the first instance, we use in-line response to provide information to the linker. (We saw this usage earlier in this chapter.) But the second instance — ah, this is what catches our attention. Here we use in-line response to define a pseudotarget that effectively causes MAKE to create the module definition file for us! This strategy assures us that two conditions — critical to our Presentation Manager application — are met:

- The module definition file is present.
- The module definition file is current.

Somewhat as an anticlimax, we conclude the makefile by defining the indirect dependencies for the object file `pmsamp.obj` and the resource file `pmsamp.res`.

Notice that the `$(TARGET:.exe=.dlg)` file is a part of the direct dependency for `$(TARGET:.exe=.res)`. This is because when the resource file is compiled, it includes all dialog files (`.dlg`) as part of the source file. Refer to the Presentation Manager sample source file (`.rc`) to see how the include process occurs. We must define the dialog file as we would any include file for a source file.

As we see from this example, no mystery attaches to using MAKE to create a GUI application. The makefiles we write to build Presentation Manager applications are very much like any other makefile. We use the same fundamentals and similar techniques. As in any development project, we need only take care that our program conform to the requirements of the environment.

The Presentation Manager example we have just discussed presents a single point of view of how best to use MAKE in a PM development effort. Refer to Borland's user documentation for more information about PM development and C++ applications. The complete source code for this example appears in Appendix G.

Summary

Two *make* utilities available on OS/2 are NMAKE from IBM and MAKE from Borland. Each version of *make* supports an extensive list of command line options. A number of these options are unique to NMAKE; others are unique to MAKE.

- Refer to page 132 for a list of NMAKE options.
- Refer to page 160 for a list of MAKE options.

Both versions of *make* also support preprocessing directives. These directives allow us to control the operation of *make* with conditional and other special-purpose statements. A preprocessing directive begins with ! in the first column of the line on which it appears. The utility processes any directives it finds in the makefile before executing the commands associated with targets.

For NMAKE:

- Two frequently used directives are `ifdef` and `if`. Any statement(s) associated with `ifdef` and `if` must be followed by an `endif` directive. Both directives can also use the `else` directive to establish alternative actions for NMAKE to take.

- The `ifdef` directive determines whether the name that follows it has been defined for the current makefile. The `if` directive evaluates the expression that follows it (`if` *expr*) and executes the statements until the `endif` or `else` when *expr* is nonzero.

- A special use of the `if` directive is to pass a command directly to `cmd.exe`. In this case, the command to be passed must be enclosed in `[ ]`.

For MAKE:

- One frequently used directive is the `if` directive. Any statement(s) associated with this directive must be followed by an `endif` directive. The `if` directive can also use both the `elif` and the `else` directives to establish alternative actions for MAKE to take.

- When followed by the name of a macro, the `if` directive tests the value assigned to that macro. For example,
```
if $(DEBUG) == 3
```

- The predefined macro `$d` can be used with the `if` directive to determine whether that macro has been defined for the current makefile; its use can be extended with the NOT operator (`!`) to test that a macro has not been defined. For example,
```
if !$d(DEBUG)
```

Both versions of *make* support the same conditional operators as the C programming language. When we use these operators with the utilities' respective conditional preprocessing directives, we can extend the power of *make* to achieve complex testing and branching activities from within the makefile.

Both versions of *make* recognize the \ character as a line continuation character. However, we may want *make* to see \ as a literal character; for example, to represent the root directory on OS/2. Each version of *make* has its own strategy for dealing with this situation.

For NMAKE:

- The caret (^) is a special control character that turns off the special meaning of other characters. When used before a backslash (^\), it tells NMAKE that the backslash is a literal character and not a line continuation indicator.

- Similarly, when used in a macro definition before a newline character, the ^ tells NMAKE that the newline is part of the definition, not an end-of-line marker for NMAKE. Another way to represent the newline character without having to use ^ is to use `\n` where the newline should occur.

For MAKE:

- Because the utility does not support escape sequences (such as ^), we use \\ to represent a literal \ character.

A complete and correct list of dependencies is critical to the successful operation of MAKE. Source files that *make* uses to build a target often contain header files (`.h`) or include files (`.inc`). These can easily be omitted from a makefile, usually with unwanted (if not disastrous) results.

For NMAKE and MAKE:

- Dependency checking can be performed dynamically as *make* executes to ensure that all dependencies are known. Refer to Appendix I for an example of a program that produces an accurate, up-to-date list of dependencies for a makefile.

For MAKE only:

- The `a` option issued on the command line allows dependency checking to be performed dynamically as MAKE executes to ensure that all dependencies are known to MAKE. An alternative technique is to use the `.autodepend` statement within the makefile to ensure dependency checking occurs every time the makefile is executed.

An input file provides input to a command; in this case, to a command within a makefile. In-line response files are closely related to input files. The difference is that we create an input file external to and independent of the makefile in which it is used. In contrast, we write an in-line response file as part of the makefile, and the file is created as *make* executes. In either case, we use the "at" character (`@`) to tell a linker to use an input file.

For NMAKE:

- The delimiters for in-line response are `<<...<<`.
- We assign a name to the temporary file that is created by in-line response. If a file of the same name already exists, it is overwritten.
- If we want to prevent NMAKE from removing the in-line response file, we must follow the closing `<<` delimiter with the `keep` command.

For MAKE:

- The `&&` notation indicates the beginning of the temporary file created by in-line response. The delimiters for in-line response can be any character that is not special to *make* (for example, `&&|...|`).
- The file created by in-line response is assigned a name that is generated automatically; optionally, we can assign a name of our choice. If a file of the same name already exists, it is overwritten.
- If we want to prevent MAKE from removing the in-line response file, we must invoke make on the command line with the `K` option.

By combining a little-known feature of *make* with the concept of batch files, we can achieve the equivalent of OS/2 command level processing from within a makefile.

Both IBM and Borland support a special file where we can store macros and inference rules that we define and want to use frequently.

- For NMAKE, the special file is called `tools.ini`.
- For MAKE, the special file is called `builtins.mak`.

Each version of *make* reads the respective special file automatically (unless we use the `r` option). Thus, these files provide a convenient device for customizing default options for *make*.

For each version of *make*, if a makefile and the special file contain conflicting definitions or rules, *make* uses the makefile in preference to the special file. In this respect, both versions of *make* follow the same precedence. However, in other respects the order of precedence varies for the two versions.

- For NMAKE the precedence of macros from highest (1) to lowest (5) is:

  1. macros on the command line
  2. macros in the makefile
  3. inherited macros (environment variables)
  4. macros in the `tools.ini` file
  5. predefined macros

  When we use the `e` option we tell NMAKE that environment variables take precedence over macro names that appear in a makefile. In this case, the order of precedence for macros (highest (1) to lowest (5)) is:

  1. macros on the command line
  2. inherited macros (environment variables)
  3. macros in the makefile
  4. macros in the `tools.ini` file
  5. predefined macros

- For MAKE the precedence of macros from highest (1) to lowest (4) is:

  1. macros in the makefile
  2. macros in the `builtins.mak` file
  3. inherited macros (environment variables)
  4. macros on the command line

  When we use the `e` option we tell MAKE that environment variables take precedence over macro names that appear in a makefile. In this case, the order of precedence for macros (highest (1) to lowest (4)) is:

  1. macros in the makefile
  2. inherited macros (environment variables)
  3. macros in the `builtins.mak` file
  4. macros on the command line

We can combine Presentation Manager (PM) tools with *make* to create graphical user interface (GUI) applications. The tools we need are PM-compiled programs; OS/2 libraries; Resource Compilers and resource files; a module definition file; and, for MAKE, an OS/2 interface program. The in-line response function of *make* allows us to create the module definition file from within the makefile so we are assured that the module definition file is always present and always current.

Chapter 7

UNIX System's MAKE

In earlier chapters we looked at concepts and strategies that, for the most part, apply universally to all versions of *make*. Here we focus our attention on the MAKE utility on UNIX systems. We assume you have read Chapters 1 through 4.

We begin this chapter by showing how to invoke MAKE. Then we explore several topics that were introduced earlier as they apply specifically to MAKE on UNIX systems. For example, we will revisit options, environment variables, predefined macros, and inference rules in the UNIX environment. New topics introduced in this chapter include the Source Code Control System (SCCS) and an example of how to develop applications for a graphical user interface (GUI) environment.

Invoking MAKE

Here is the format for invoking MAKE on the command line:

```
make  [options]  [macro_definitions]  [targets]
```

The brackets (`[ ]`) around a field name indicate that the field is optional. Thus, all three fields are optional for MAKE.

Command Line Options

MAKE supports an extensive list of command line options. If you are familiar with other versions of *make*, some of these options may look familiar. However, a number of MAKE options differ from those supported in other versions of *make*.

MAKE is case sensitive with respect to options; that is, MAKE treats `s` and `S` as two different options. Remember that when we specify an option on the command line, we must precede the option letter with a dash (-).

Here are some of the options that MAKE supports:

`d`	Debug mode: print detailed information about the files being examined
`e`	Let environment variables override macro definitions
`f` *filename*	Specify makefile to be *filename*
`i`	Ignore exit/error codes
`n`	Display commands but do not execute them
`p`	Print values of defined macros and pseudotargets
`q`	Return exit code
`r`	Do not use built-in rules
`s`	Suppress command display while executing
`t`	Change target modification dates

Makefile Command Lines

A target can have any number of command lines. For MAKE on UNIX systems, a command line in a makefile must begin with a tab character. (In this respect, this version of *make* differs from the versions of *make* that run on PC systems, where command lines may begin with either a tab or a space character.)

MAKE starts a separate process for each command line in the makefile. Most of the time this characteristic does not affect the execution of the

commands. There is one case, however, that we need to be aware of. Let's consider a makefile with a target that contains the two command lines:

```
cd  somedir
ls
```

The first command tells MAKE to change to a new directory called `somedir`. The second command, `ls`, tells MAKE to list the files in the current directory. However, MAKE runs a separate process to execute this command. Because each command line runs as a separate process, the `cd` command does not affect subsequent lines. That is, in the UNIX system, MAKE issues `cd somedir` in a separate shell and waits for its completion. This separate shell does not affect the current shell where MAKE issues the `ls` command. Consequently, the `ls` command refers to the files in the current directory, *not* the files in `somedir`.

How can we get around this apparent limitation? If we replace the two command lines shown above with the single command line

```
cd  somedir;  ls
```

MAKE runs one process that consists of two commands: `cd  somedir` and `ls`. Now the `ls` command refers to the files in `somedir`. After the process for this command line terminates, MAKE is back in the directory where it was before it executed the `cd somedir` command.

Another way to indicate that a command continues on the next line (and, therefore, is part of the same process) is to use the backslash character (\):

```
cd somedir; \
ls
```

Note that the UNIX system still requires the semicolon (;) between multiple commands to be executed in the same process, regardless of whether the commands are on one physical line in the makefile or on several lines with \ line continuation indicators.

Environment Variables

MAKE has access to environment variables. We can create environment variables for the Bourne shell or the Korn shell at the shell prompt ($) using:

```
$ TARGET=prog
$ export TARGET
```

In the C shell, we use instead:

```
% set TARGET prog
```

If our makefile is

```
OBJS   = prog1.o prog2.o

$(TARGET): $(OBJS)
        $(CC) -o $@ $(OBJS)
```

and `TARGET` is not defined in the makefile, MAKE uses the value of the environment variable `TARGET`. What if `TARGET` is not defined as an environment variable? Then MAKE reports a syntax error:

```
Make: line 3: syntax error.  Stop.
```

MAKE reads the environment variables before it reads the file that contains the macros, targets, and commands (in UNIX systems, the default file MAKE uses is either `makefile` or `Makefile`). Therefore, we can override an environment variable by assigning a new value to a macro with the same name in the makefile.

If we modify the makefile to include a definition for `TARGET`

```
OBJS   = prog1.o prog2.o
TARGET = prog

$(TARGET): $(OBJS)
        $(CC) -o $@ $(OBJS)
```

MAKE will use the value of `TARGET` in the makefile even when `TARGET` is an environment variable. What if we want to use the value of the environment variable `TARGET` instead of the value of `TARGET` that appears in the makefile? It is easy. We simply invoke MAKE with the `e` option. Assume our makefile is called `mk`. We tell MAKE the name of the file using the `f` option along with the `e` option:

```
make -e -f mk
```

MAKE uses the environment variable `MAKEFLAGS` to remember input options. `MAKEFLAGS` remembers input options through the lifetime of the current `make` command. This means that when we use MAKE recursively (as we discussed in Chapter 4), any option that we specify in the initial invocation of MAKE remains in effect via the `MAKEFLAGS` variable for all subsequent instances of MAKE.

For example, when we invoke MAKE with the `n` option, we tell MAKE to show the commands that would be executed, but without executing them. This is true, with one exception: when MAKE finds the `MAKE` macro in a makefile, MAKE reinvokes itself and continues to check and report the lines that should be executed for the second instance of MAKE. This behavior of `n` violates the general rule that MAKE does not *execute* the commands. The reason for this exception is that MAKE needs to follow a recursion or a hierarchy of makefiles to determine all the commands that will be executed when we invoke MAKE without the `n` option.

Macros and Macro Precedence

The MAKE utility in UNIX systems supports more predefined macros than the PC versions of *make*. Remember the `p` option? When we invoke MAKE with this option

```
make -p
```

MAKE displays all predefined macros, inference rules, and environment variables (even when we do not have a makefile present). Try it on your system.

Here is a subset of the predefined macros in our UNIX system:

```
ASFLAGS =
AS      =  as
CFLAGS  =  -O
CC      =  cc
LDFLAGS =
LD      =  ld
LFLAGS  =
LEX     =  lex
YFLAGS  =
```

```
YACC    =  yacc
MAKE    =  make
```

We chose to show you these predefined macros because they are similar to those in the PC versions of *make*. Four of these predefined macros may be unfamiliar to you: `LEX` and `YACC` and the associated flags macros are unique to the UNIX system's MAKE. These macros are defined to invoke the `lex` and `yacc` tools that are available on UNIX systems.

MAKE also has access to internal macros. Some of these macros correspond to the macros available in PC versions of *make*. Some of these macros are unique to MAKE in the UNIX system. Here are some examples showing how internal macros are used in UNIX systems.

`$*` is the name of the current target without an extension. For example in,

```
prog1.o:  prog1.c
          $(CC)  $(CFLAGS)  $*.c
```

the value of `$*` is `prog1`. `$*` is commonly used in inference rules.

`$@` is the full name of the current target. For example, in

```
prog:     $(OBJS)
          $(CC)  -o  $@  $*.c
```

the value of `$@` is `prog`.

`$<` is the module that is out of date with respect to the target. For example,

```
.c.o:
          $(CC)  $(CFLAGS)  $<
```

Notice that `$<`, in an inference rule such as `.c.o`, is equivalent to `$*.c`:

```
.c.o:
        $(CC)  $(CFLAGS)  $*.c
```

The internal macro `$<` can be used only in inference rules.

`$?` is the list of modules that are newer than the target. In other words, `$?` is the list of modules that MAKE will operate on in order to build the current target. For example, in

```
OBJS = prog1.o  prog2.o  prog3.o

prog:  $(OBJS)
       echo  $?
```

If the files `prog2.o` and `prog3.o` are newer than the target `prog`, the `echo` command displays

```
prog2.o  prog3.o
```

There are many more macros. Use the `p` option with `make` and consult your UNIX system documentation to determine which ones are available on your system.

On UNIX systems, a macro defined on the command line has precedence over all other macros in MAKE. At the next level of precedence are macros defined in the makefile. A macro defined in the makefile has precedence over environment variables and predefined macros. Finally, an environment variable has precedence over predefined macros. (See also "Environment Variables" on page 195.)

For example, if a macro name appears in the environment, then the UNIX system's MAKE uses the value of the environment variable as long as a macro with the same name does not appear either on the command line or in the makefile.

Let's consider a specific case. `CFLAGS` is a predefined macro that we discussed in Chapter 3. If `CFLAGS` does not appear on the command line, in the makefile, or in the environment, then MAKE uses its predefined value. However, if we redefine the value of `CFLAGS` at any of these levels, MAKE

ignores the predefined value and uses the new definition. This is because predefined macros have the lowest precedence in MAKE.

Suffixes and Inference Rules

In the UNIX systems we have used, the *make* utility knows more suffixes and has more built-in inference rules than the PC versions of *make*. When we run

```
make -p
```

MAKE lists predefined macros, suffixes, and built-in inference rules.

The list of suffixes that MAKE knows is rather long. Here is the predefined pseudotarget `.SUFFIXES:` as it is defined on one UNIX system:

```
.SUFFIXES: .o  .c  .c~  .y  .y~  .l  .l~  .s  .s~   \
        .sh  .sh~  .h  .h~  .out  .F  .f  .e  .r     \
        .yr  .ye  .cl  .p
```

The extensions followed by ~ (for example, `.c~`) refer to Source Code Control System (SCCS) files. We address SCCS in the next section. Consult your UNIX system documentation for an explanation of any extensions that may be unfamiliar to you. (Note that `.SUFFIXES:` does not contain continuation lines; we've added them here to split a long line.)

Here are some of the built-in inference rules that MAKE knows:

```
.c.o:
        $(CC)  $(CFLAGS)  -c  $<

.s.o:
        $(AS)  $(ASFLAGS)  -o  $@  $<

.c:
        $(CC)  $(CFLAGS)  $(LDFLAGS)  $<  -o  $@
```

Let's examine the inference rules a little more closely. Recall that UNIX systems use `.o` for object files (in contrast to DOS and OS/2, which use the `.obj` extension). The command line associated with the `.c.o:` inference rule is:

```
$(CC)  $(CFLAGS)  -c  $<
```

This command line indicates that MAKE will invoke the C compiler with the value of `CFLAGS` and the `-c` compiler option to compile the current target (represented by the predefined macro `$<`) to obtain an object file. The `.s.o:` inference rule similarly applies to building object files from assembly language files.

The last inference rule listed above

```
.c:
        $(CC)  $(CFLAGS)  $(LDFLAGS)  $<  -o  $@
```

does not look like any of the inference rules we've seen so far. Most noticeably it contains only one extension (`.c:`). This is called a *null suffix* because MAKE assumes an empty (null) suffix between the `.c` extension and the colon (`:`). This is the rule MAKE uses to build an executable file from a `.c` file. (Remember that UNIX systems do not have a specific extension for executable files, unlike DOS and OS/2, which use either the `.exe` or the `.com` extension for executable files.)

The benefit we derive from the `.c:` inference rule is that we do not have to create a makefile to compile a program that appears in one source file. For example, if we have a file called `prog.c` that contains a program and its associated functions, we can issue the command

```
make  prog
```

even though we have not created a makefile with the `prog:` target. When MAKE executes this command, it applies the built-in `.c:` inference rule, compiles `prog.c`, and produces an executable file `prog`. (We are assuming, of course, that `prog.c` does not contain any syntax errors.) This is a quick and easy way to produce an executable file from a single source file.

When you use the `p` option with MAKE, you will see more inference rules. Try the `p` option on your system and consult your UNIX system documentation.

About SCCS

The Source Code Control System (SCCS) is a collection of commands that help control changes that we make to files, how the files are stored, and how

we retrieve them. Let's say we use two files in our project, `prog1.c` and `prog2.c`. The file `prog1.c` contains a call to a function:

```
void f(void);

void main()
{
    f();
}
```

The function `f()`, which appears in `prog2.c`, displays a message:

```
#include    <stdio.h>

void f(void)
{
    printf("\tin function f()\n");
}
```

The makefile is familiar because we are using plain `.c` source files:

```
prog:           prog1.o  prog2.o
                $(CC)  -o  prog  *.o

prog1.o:        prog1.c

prog2.o:        prog2.c
```

To turn the files over to SCCS we use the `admin` command. For example, in

```
admin  -iprog1.c  s.prog1.c
```

The `i` option identifies the file that is the input to SCCS, `prog1.c`. The name `s.prog1.c` is the name that SCCS will use for the file. All SCCS files must begin with the `s.` prefix. Similarly:

```
admin  -iprog2.c  s.prog2.c
```

At this point it is advisable to remove the original files (`prog1.c` and `prog2.c`) to ensure that we access only the versions of the files that SCCS controls:

```
rm  prog1.c  prog2.c
```

We may retrieve a copy of a file from SCCS at any time. To do this, we use:

```
get  s.prog1.c
get  s.prog2.c
```

Then we run `make` to build `prog`.

This is acceptable for a couple of files, but it becomes tedious and error prone when we have many files that must be retrieved from SCCS. An alternative is to modify the makefile to access the files in SCCS directly:

```
.c~.o:
                get  $<
                $(CC)  -c  $*.c
                rm  -f  $*.c

prog:           prog1.o  prog2.o
                $(CC)  *.o  -o  prog

prog1.o:        s.prog1.c

prog2.o:        s.prog2.c
```

We indicate that `prog1.o` and `prog2.o` depend on the source files under the control of SCCS, `s.prog1.c` and `s.prog2.c`, because otherwise MAKE will look for `prog1.c` and `prog2.c` instead of the SCCS files. We also added an inference rule:

```
.c~.o:
                get  $<
                $(CC)  -c  $*.c
                rm  -f  $*.c
```

MAKE uses `.c~` to indicate a C source file under SCCS control. MAKE does not have a way to indicate "a file that begins with `s.` prefix." Because inference rules work with file extensions (and not prefixes), the developers of MAKE used the tilde character (`~`) to identify SCCS files. The command

```
get  $<
```

retrieves a copy of the source file from the appropriate SCCS file (`prog1.c` from `s.prog1.c`, for example). The next line in the inference rule is familiar: it tells *make* to generate the object (`.o`) file by compiling the source file. Next we remove the copy of the source file that we obtained through the `get` command; we do not wish to keep a copy of the source around. Every time we need a copy of the source, we should use the `get` command because the file may have been modified since the last time we obtained a copy of it.

Interestingly, MAKE already has rules involving SCCS. If we run

```
make  -p
```

on a UNIX system, we see the built-in rules for source files under SCCS control. Here are two of those built-in rules:

```
.c~.o:
        $(GET)  $(GFLAGS)  -p  $<  > $*.c
        $(CC)  $(CFLAGS)  -c  $*.c
        -$(RM)  -f  $*.c

.s~.o:
        $(GET) $(GFLAGS) -p $< > $*.s
        $(AS) $(ASFLAGS) -o $*.o $*.s
        -$(RM) -f $*.s
```

The `p` option in the first command line tells the `$(GET)` command to write its output (i.e., the contents of the file) to standard output; then `> $*.c` routes that output to a file with `.c` extension. Thus, if we rewrite the makefile to read

```
prog:           prog1.o  prog2.o
                $(CC)  *.o  -o  prog

prog1.o:        s.prog1.c

prog2.o:        s.prog2.c
```

we obtain the same behavior as with the previous makefile that contained the `.c~.o` rule.

What if we want to modify a file that exists in SCCS? The `e` option in `get` allows us to get a copy of the file that we can modify. For example, if we want to modify `prog1.c`, we must use the `get` command to request a copy of the file suitable for editing:

```
get  -e  s.prog1.c
```

We then add our modifications to the *modifiable* version of `prog1.c`.

```
#include    <stdio.h>
void f(void);

void main()
{
    printf("before f()\n");
    f();
    printf("after f()\n");
}
```

A copy of a file is *modifiable* only when we obtain it with the `e` option to the SCCS `get` command. After we make the necessary modifications and test `prog1.c`, we can put the modifications back into SCCS using the `delta` command:

```
delta  s.prog1.c
```

The `delta` command accepts *only* files that were obtained using `get` with the `e` option. The next time we run MAKE, we will get the current version of `prog1.c`; that is, the version of the file with our modifications.

SCCS maintains a list of the lines that change from one version of the file to the next. Consequently, we do not double the size of the file every time we make a change. We are not storing a new copy of the file; we are storing only a copy of the lines that were modified after the last `get -e` command.

SCCS maintains information about source files. Every time we request a copy of a file from SCCS, we obtain the current version of the file. This is true whether we request the file directly, using the `get` command, or indirectly, using MAKE's built-in inference rules.

SCCS also allows us to request an earlier version of the file by specifying a version number. Most of the time, however, we are interested only in the

current version of the file; consequently, we will conclude our discussion of SCCS at this point. For more information about using SCCS, refer to your UNIX documentation.

Using MAKE to Develop GUI Applications

More and more applications are being developed for graphical user interfaces (GUIs) such as X Windows. As we demonstrate in this section, programmers can use X Windows libraries with MAKE to create GUI applications.

If we wish to develop an X Windows application, we must have:

- X Windows compiled program(s)
- X Windows libraries — `libXm.lib`, `libXmu.lib`, `libXt.lib`, `libXext.lib`, and `libX11.lib`.

You can find detailed information about these items in the X Windows reference manuals for your UNIX system.

The following makefile (`xwindows.mak`) creates an X Window with text and two pull-down menus. One accesses the dialog feature, and the other exits the program.

```
#
# Makefile for UNIX X Window System
# (with Motif Widgets) Example
#
# Mastering Make: A Guide to Building Programs on
# DOS, OS/2, and UNIX* Systems
# Tondo, Nathanson, Yount
#
# Prentice Hall (C) 1994
#

RM              = rm -f

CFLAGS          = -g  -O

EXTENSIONLIB    = -lXext
XLIB            = $(EXTENSIONLIB) -lX11
XMLIB           = -lXm
```

```
XMULIB          = -lXmu
XTOOLLIB        = -lXt

LOCAL_LIBRARIES = $(XMLIB) $(XMULIB) $(XTOOLLIB) $(XLIB)

OBJS            = xmsamp.o

xmsamp: $(OBJS)
                $(RM) $@
                $(CC) -o $@ $(OBJS) $(LOCAL_LIBRARIES)
```

This X Windows example presents a single point of view of how best to use MAKE in an X Windows development effort. Refer to your UNIX system documentation for more information about X Windows development and C applications.

The complete source code for this example appears in Appendix H.

Summary

The MAKE utility on UNIX systems supports an extensive list of command line options. A number of these options are unique to MAKE. Refer to page 194 for a list of common options. The list of options supported by your system may be much longer.

MAKE runs a separate process for each command line in the makefile. Therefore, if we want to change directories and issue a command to take effect in the new directory, we must remember to place both commands on a single logical command line in the makefile and to separate the commands with a semicolon (;).

Because MAKE reads environment variables before reading the makefile, we can redefine an environment variable by assigning a new value to it in the makefile. The environment variable `MAKEFLAGS` remembers input options; MAKE uses the options stored in `MAKEFLAGS` when we apply the recursion process to MAKE.

From highest (1) to lowest (4), the order of precedence for macros in MAKE is:

1. macros defined on the command line
2. macros defined in the makefile
3. macros defined in the system environment
4. macros predefined by the system

When we use the `e` option, we tell MAKE that environment variables take precedence over matching macro names that appear in the makefile (see "Environment Variables" on page 195). In this case, the order of precedence for macros (highest (1) to lowest (4)) is:

1. macros defined on the command line
2. macros defined in the system environment
3. macros defined in the makefile
4. macros predefined by the system

The UNIX system's MAKE supports more predefined macros, knows more suffixes, and has more inference rules than the PC versions of *make*. Two predefined macros unique to MAKE on UNIX systems are `LEX` and `YACC`. Also unique is the null suffix exemplified by the `.c:` inference rule, which MAKE uses to build an executable file from a `.c` file.

We can display the values of all predefined macros, inference rules, and environment variables known to MAKE by using the `p` option.

The Source Code Control System (SCCS) is a collection of commands that help control changes that we make to files, how the files are stored, and how we retrieve them. Here are some SCCS commands we use to maintain source files:

- `admin` to turn files over to SCCS
- `get` to retrieve a file from SCCS
- `delta` to give a file back to SCCS after having issued a `get` with an `e` option (for editing)

MAKE uses the tilde character (`~`) after file extensions to identify SCCS files. For example, the predefined inference rule `.c~.o:` has the commands necessary to produce an object file (`.o`) from a C source file that is maintained by SCCS (`.c~`).

MAKE and SCCS form a powerful combination for maintaining versions of source files for large projects.

Appendix A

Comparing MAKE

Feature	Microsoft's/IBM's NMAKE	Borland's MAKE	UNIX System's MAKE
Name	`nmake`	`make`	`make`
Macros			
user-defined	yes	yes	yes
predefined	yes	yes	yes
Inference rules			
user-defined	yes	yes	yes
predefined	yes	yes	yes
Pseudotargets			
user-defined	yes	yes	yes
predefined	yes	yes[a]	yes
In-line files	yes	yes	no
Directives	yes	yes	no
Information file	`tools.ini`	`builtins.mak`	—
Precede commands	with space(s) or tab(s)		with tab(s)

[a] Predefined pseudotargets are called *dot directives* in Borland's MAKE.

Macro	Description	Microsoft's/ IBM's NMAKE	Borland's MAKE	UNIX System's MAKE
`$*`	Target with path, without extension	yes	yes	yes
`$@`	Target with path, with extension	yes	yes	yes
`$<`	File name with path that is out of date with respect to the target; can be used only in inference rules	yes	yes	yes
`$**`	List of dependent files	yes	yes	yes
`$?`	File names with path that are out of date with respect to the target	yes	yes	yes
`$$@`	Target file that is currently being evaluated; use only in a dependency list	yes	no	no
`$&`	Target without path, without extension	no	yes	no
`$.`	Target without path, with extension	no	yes	no
`$:`	Path for dependent file	no	yes	no

Appendix B

Microsoft's NMAKE

This is a summary of the built-in items for NMAKE on DOS that are used and described in this book. Refer to the documentation from Microsoft for more details and additions to these lists.

Command Line

```
nmake  [options]  [macros]  [targets]
```

- Arguments enclosed in `[ ]` are optional
- The command line entry is case sensitive

Options

`?`	Display short help
`a`	Build all targets
`b`	Execute a dependency even if time stamps are equal
`c`	Suppress messages
`d`	Display modification dates
`e`	Let environment variables override macro definitions
`f` *filename*	Specify makefile to be *filename*
`help`	Display help information

`i`	Ignore exit/error codes
`k`	Build unrelated parts of dependency tree if command fails
`m`	Swap NMAKE to disk to save memory space
`n`	Display commands but do not execute them
`nologo`	Suppress copyright message
`p`	Print values of defined macros and pseudotargets
`q`	Return exit code
`r`	Ignore `tools.ini` file
`s`	Suppress command display while executing
`t`	Change target modification dates
`v`	Cause macros to be inherited in NMAKE recursion
`x` *filename*	Specify destination for error messages
`z`	Internal to Programmer's Work Bench

- Precede options with `-` or `/`
- Options are not case sensitive
- *filename* arguments (to options `f` and `x`) are not case sensitive

Predefined Macros

`$*`	base filename with path
`$@`	current target
`$<`	full filename with path
`$**`	list of dependent files
`$?`	files out of date
`$$@`	target file being evaluated
`CC`	C compiler (`cl`)
`CFLAGS`	empty macro
`AS`	assembler (`masm`)
`AFLAGS`	empty macro

`MAKE`	Microsoft's NMAKE command (`nmake`)

- All predefined and user-defined macros are case sensitive

Predefined Inference Rules

```
.asm.obj:
        $(AS)  $(AFLAGS)  $*.asm ;

.c.obj:
        $(CC)  $(CFLAGS)  /c  $*.c

.cpp.obj:
        $(CC)  $(CPPFLAGS)  /c  $*.cpp

.c.exe:
        $(CC)  $(CFLAGS)  $*.c

.rc.res:
        $(RC)  $(RFLAGS)  /r  $*
```

Predefined Pseudotargets

```
.SILENT:
.IGNORE:
.SUFFIXES:  .asm  .c  .cpp  .obj  .exe
```

- All predefined pseudotargets are case sensitive; user-defined pseudotargets are not

Preprocessing Directives

`cmdswitches`	`error` *message*	`ifndef` *name*
`else`	`if defined` *name*	`include <`*filename*`>`
`elseif` *expr*	`if exist` *path*	`include "`*filename*`"`
`elseifdef` *name*	`if` *expr*	`message` *text*
`elseifndef` *name*	`ifdef` *name*	`undef` *name*
`endif`		

- Precede with ! (left justified; no spaces or tabs)
- Directives are not case sensitive

Special Characters

comment	#
escape control character	^
in-line file delimiters	<<*filename* ⋮ <<
or	
	<<*filename* ⋮ <<keep
or	
	<<*filename* ⋮ <<nokeep
line continuation indicator	\
response file indicator	@*filename*

- The ending << and <<nokeep are equivalent

Appendix C

IBM's NMAKE

This is a summary of the built-in items for NMAKE on OS/2 that are used and described in this book. Refer to the documentation from IBM for more details and additions to these lists.

Command Line

```
nmake  [options]  [macros]  [targets]
```

- Arguments enclosed in `[ ]` are optional
- The command line entry is case sensitive

Options

`?`	Display short help
`a`	Build all targets
`c`	Suppress messages
`d`	Display modification dates
`e`	Let environment variables override macro definitions
`f` *filename*	Specify makefile to be *filename*
`help`	Display help information
`i`	Ignore exit/error codes

`n`	Display commands but do not execute them
`nologo`	Suppress copyright message
`p`	Print values of defined macros and pseudotargets
`q`	Return exit code
`r`	Ignore `tools.ini` file
`s`	Suppress command display while executing
`t`	Change target modification dates
`x` *filename*	Specify destination for error messages

- Precede options with `-` or `/`
- Options are not case sensitive
- *filename* arguments (to options `f` and `x`) are not case sensitive

Predefined Macros

`$*`	base filename with path
`$@`	current target
`$<`	full filename with path
`$**`	list of dependent files
`$?`	files out of date
`$$@`	target file being evaluated
`CC`	C compiler (`icc`)
`CFLAGS`	empty macro
`AS`	assembler (`masm`)
`AFLAGS`	empty macro
`MAKE`	IBM's NMAKE command (`nmake`)

- All predefined and user-defined macros are case sensitive

Predefined Inference Rules

```
.asm.obj:
        $(AS)  $(AFLAGS)  $*.asm ;

.c.obj:
        $(CC)  $(CFLAGS)  /c  $*.c

.cpp.obj:
        $(CC)  $(CFLAGS)  /c  $*.cpp

.c.exe:
        $(CC)  $(CFLAGS)  $*.c

.rc.res:
        $(RC)  $(RFLAGS)  /r  $*
```

Predefined Pseudotargets

```
.SILENT:
.IGNORE:
.SUFFIXES:  .asm  .c  .cpp  .obj  .exe
```

- All predefined pseudotargets are case sensitive; user-defined pseudotargets are not

Preprocessing Directives

`cmdswitches`	`if` *expr*	`include <`*filename*`>`
`else`	`ifdef` *name*	`include "`*filename*`"`
`endif`	`ifndef` *name*	`undef` *name*
`error` *message*		

- Precede with ! (left justified; no spaces or tabs)
- Directives are not case sensitive

Special Characters

comment	`#`
escape control character	`^`
in-line file delimiters	`<<`*filename* ⋮ `<<`
or	
	`<<`*filename* ⋮ `<<keep`
or	
	`<<`*filename* ⋮ `<<nokeep`
line continuation indicator	`\`
response file indicator	`@`*filename*

- The ending `<<` and `<<nokeep` are equivalent

Appendix D

Borland's MAKE

This is a summary of the built-in items for MAKE on DOS and OS/2 that are used and described in this book. Refer to the documentation from Borland for more details and additions to these lists.

Command Line

```
make  [options]  [targets]
```

- Arguments enclosed in `[ ]` are optional
- The command line entry is case sensitive

Options

`?`	Display help information
`a`	Perform automatic dependency check on all object files
`B`	Build all targets regardless of time stamp (file dates)
`d`*directory*	Identify where to write the swap file; *directory* can include a drive letter
`D`*identifier*	Define the value of *identifier* to be 1
`D`*identifier*`=`*string*	Define the value of *identifier* to be *string*

`e`	Let environment variables override macro definitions
`f`*filename*	Specify the makefile to be *filename*; space between `f` and the filename is optional (`f` *filename*)
`h`	Display help information
`i`	Ignore exit/error codes
`I`*directory*	Search for include files in *directory*
`K`	Retain any temporary file(s) created by MAKE
`m`	Display time and date stamp for each file as the file is being processed
`n`	Display commands but do not execute them
`N`	Change the behavior of in-line response processing to be compatible with NMAKE from Microsoft
`p`	Display all macros and inference rules before processing the makefile
`r`	Ignore rules that are defined in the `builtins.mak` file
`s`	Suppress command display while executing
`S`	Swap MAKE out of memory while executing commands
`U`*identifier*	Undefine *identifier* if previously defined
`W`	Write the current specified nonstring options to `make.exe`

- Precede options with `-` or `/`
- No space between option and argument, except `f`
- Options and option arguments are case sensitive, except option arguments *filename* and *directory* are not case sensitive

Predefined Macros

`$*`	base filename with path
`$<`	full filename with path
`$@`	full filename with path
`$**`	list of dependent files
`$?`	files out of date
`$$@`	target file being evaluated
`$&`	target without path, without extension
`$.`	target without path, with extension
`$:`	path for dependent file
`CC`	C compiler (`bcc`)
`CFLAGS`	empty macro
`AS`	assembler (`tasm`)
`AFLAGS`	empty macro
`MAKE`	Borland's MAKE command (`make`)

Predefined Inference Rules

```
.asm.obj:
        $(AS)  $(AFLAGS)  $&.asm

.c.obj:
        $(CC)  $(CFLAGS)  /c  $&.c

.cpp.obj:
        $(CC)  $(CPPFLAGS)  /c  $&.cpp

.c.exe:
        $(CC)  $(CFLAGS)  $&.c

.rc.res:
        $(RC)  $(RFLAGS)  /r  $&
```

Dot Directives (Predefined Pseudotargets)

```
.autodepend:        .precious:        .suffixes:
.ignore:            .silent:
```

- All predefined pseudotargets are case sensitive; user-defined pseudotargets are not

Preprocessing Directives

`elif` *expr*	`error` *message*	`include "`*filename*`"`
`else`	`if` *expr*	`undef` *name*
`endif`	`include <`*filename*`>`	

- Precede with `!` (left justified; no spaces or tabs)
- Directives are not case sensitive
- Arguments to directives are case sensitive

Special Characters

comment	`#`
in-line file delimiters	`@&&|`
	⋮
	`|`
line continuation indicator	`\`
response file indicator	`@`*filename*

Appendix E

UNIX System's MAKE

This is a summary of the built-in items for MAKE on UNIX systems that are used and described in this book. Refer to the online manual page for MAKE for more details and additions to these lists.

Command Line

```
make [options] [macros] [targets]
```

- Arguments enclosed in `[ ]` are optional
- The command line entry is case sensitive

Options

`d`	Debug mode: print detailed information about the files being examined
`e`	Let environment variables override macro definitions
`f` *filename*	Specify makefile to be *filename*
`i`	Ignore exit/error codes
`n`	Display commands but do not execute them
`p`	Print values of defined macros and pseudotargets
`q`	Return exit code

r	Do not use built-in rules
s	Suppress command display while executing
t	Change target modification dates

- Precede options with -
- Options and option arguments are case sensitive

Predefined Macros

$*	base filename with path
$@	current target
$<	full filename with path
$?	files out of date
$%	file name when target is an archive
CC	C compiler (cc)
CFLAGS	empty macro
AS	assembler (as)
ASFLAGS	empty macro
MAKE	MAKE command (make)

- All predefined and user-defined macros are case sensitive

Predefined Inference Rules

```
.s~.o:
        $(GET) $(GFLAGS) -p $< > $*.s
        $(AS) $(ASFLAGS) -o $*.o $*.s
        -$(RM) -f $*.s

.s.o:
        $(AS) $(ASFLAGS) -o $@ $<

.c~.c:
        $(GET) $(GFLAGS) -p $< > $*.c
```

```
.c~.o:
        $(GET) $(GFLAGS) -p $< > $*.c
        $(CC) $(CFLAGS) -c $*.c
        -$(RM) -f $*.c

.c.o:
        $(CC) $(CFLAGS) -c $<

.c:
        $(CC) $(CFLAGS) $(LDFLAGS) $< -o $@
```

Predefined Pseudotargets

```
.SILENT:
.IGNORE:
.SUFFIXES: .o  .c  .c~  .y  .y~  .l  .l~  .s  .s~   \
       .sh  .sh~  .h  .h~  .out  .F  .f  .e  .r     \
       .yr  .ye  .cl  .p
```

- All predefined and user-defined pseudotargets are case sensitive

Preprocessing Directives

- None

Special Characters

comment	#
line continuation indicator	\

Appendix F

Windows Sample

The Program

```
//
//  Program Name: winsamp.c
//
//  Description:
//    This is a Windows sample that displays 'mastering make'
//    on the screen.
//
//  Mastering Make - Tondo, Nathanson, Yount
//  Prentice Hall (C) 1994
//


// Include Headers
#include    <windows.h>
#include    <string.h>

// Define Symbolic Constants
#define     ID_EXIT     100
#define     INFODLG     200
#define     ID_INFO     300

//  Declare Function Prototypes
int  PASCAL  WinMain          (HANDLE instance,
                               HANDLE prevInstance,
```

```
                                    LPSTR cmdLine,
                                    int cmdShow);
BOOL          InitApplication   (HANDLE instance);
BOOL          InitInstance      (HANDLE instance,
                                    int cmdShow);
long WINAPI   SetWindProc       (HWND hwnd,
                                    UINT message,
                                    WPARAM wparam,
                                    LPARAM lparam);
BOOL WINAPI   DisplayInfo       (HWND hwnd,
                                    UINT message,
                                    WORD wparam,
                                    LONG lparam);

//  Global Variables

HANDLE   HInst;                  // Window Instance Handle

// WinMain -
//   This function calls all the initialization functions
//   and starts a message queue for the Windows engine.
//   Upon receiving a PostQuitMessage it will destroy the
//   message queue.

int PASCAL  WinMain(HANDLE instance, HANDLE prevInstance,
                    LPSTR cmdLine, int cmdShow)
{

   // Define Local Variables

   MSG msg;

   // Check to see if the previous instance is NULL.
   // If the previous instance is NULL, then
   // initialize the window class information.

   if ( prevInstance == NULL )
   {
      if ( InitApplication(instance) == FALSE )
```

```
      {
         return FALSE;
      }
   }

   // Create the window and check the return code

   if ( InitInstance(instance, cmdShow) == FALSE )
   {
      return FALSE;
   }

   // Start examining the message queue

   while ( GetMessage(&msg, NULL, NULL, NULL) )
   {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }

   return msg.wParam;
}

// InitApplication -
//   This function loads the window class information
//   into the window class structure.

BOOL InitApplication(HANDLE instance)
{

   // Define local variables

   WNDCLASS  wc;

   // Load window class information structure

   wc.style         = NULL;
   wc.lpfnWndProc   = SetWindProc;
   wc.cbClsExtra    = 0;
```

```
    wc.cbWndExtra    = 0;
    wc.hInstance     = instance;
    wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName  =  "SampleMenu";
    wc.lpszClassName = "WindowSampleClassOutput";

    // Return window class information

    return RegisterClass(&wc) ;

}

// InitInstance -
//   This function creates the window and displays
//   it on the screen.

BOOL InitInstance(HANDLE instance, int cmdShow)
{

    // Define Local Variables

    HWND            hwnd;

    HInst = instance;

    // Create window

    hwnd = CreateWindow("WindowSampleClassOutput",
                        "Mastering Make Windows Sample",
                        WS_OVERLAPPEDWINDOW,
                        0,
                        0,
                        GetSystemMetrics(SM_CXSCREEN),
                        GetSystemMetrics(SM_CYSCREEN),
                        NULL,
                        NULL,
                        instance,
```

```
                        NULL);

   // Check the results

   if ( hwnd == NULL )
   {
      return FALSE;
   }

   // Show the window on the screen

   ShowWindow(hwnd, cmdShow);
   UpdateWindow(hwnd);

   return TRUE;

}

#define         MAXTEXT         67

// SetWindProc -
//   This function is the main window processing
//   function that will handle all the messages
//   by the Windows graphic engine.

long WINAPI SetWindProc(HWND hwnd, UINT message,
                  WPARAM wparam, LPARAM lparam)
{

   FARPROC        procInst;
   HDC            hdc;
   PAINTSTRUCT    ps;

   static   char  text1[MAXTEXT],
                  text2[MAXTEXT];


   switch (message)
   {
```

```
case WM_CREATE:

   // Copy the text information into the fields

   strncpy(text1, "\0", sizeof(text1));
   strncpy(text2, "\0", sizeof(text2));
   strcpy(text1,
     "Mastering Make: A Guide to Build Programs on DOS"
     "and UNIX* Systems");
   strcpy(text2,
     "              and will now include WINDOWS and "
     "OS/2 Support");
   break;
case WM_PAINT:

   // Set up a display context to begin painting

   hdc = BeginPaint (hwnd, &ps);

   TextOut(hdc, 85, 100,  text1, strlen(text1));
   TextOut(hdc, 85, 120,  text2, strlen(text2));

   // Tell Windows you are done painting

   EndPaint (hwnd,  &ps);
   break;
case WM_COMMAND:
   if ( wparam == ID_EXIT )
   {
      PostQuitMessage(0);
   }

   if ( wparam == ID_INFO )
   {
      procInst = MakeProcInstance(DisplayInfo, HInst);

          DialogBox(HInst,
                    "InfoDlg",
                    hwnd,
```

```
                        procInst);

                FreeProcInstance(procInst);
            }
            break;
        default:
            return DefWindowProc(hwnd, message, wparam, lparam);
    }

    return NULL;

}

// DisplayInfo -
//   This function displays the info dialog
//   to the screen.

BOOL WINAPI DisplayInfo(HWND hwnd, UINT message,
                    WORD wparam, LONG lparam)
{

    // Check the message

    switch ( message )
    {
        case WM_INITDIALOG:
            return TRUE;
            break;
        case WM_COMMAND:
            EndDialog(hwnd, TRUE);
            return TRUE;
    }

    return FALSE;

}
```

The .rc File

```
//
// Mastering Make - Sample Windows Resource File
//
// Mastering Make - Tondo, Nathanson, Yount
// Prentice Hall (C) 1994
//

#include    "windows.h"

#define     ID_EXIT     100
#define     INFODLG     200
#define     ID_INFO     300

SampleMenu MENU
BEGIN
  POPUP         "&Info"
    BEGIN
        MENUITEM "&Info", ID_INFO
    END

  POPUP         "&Exit"
    BEGIN
        MENUITEM "&Exit", ID_EXIT
    END
END

#include  "winsamp.dlg"
```

The .dlg File

```
//
// Mastering Make - Windows Sample Dialog File
//
// Mastering Make - Tondo, Nathanson, Yount
// Prentice Hall (C) 1994

InfoDlg DIALOG 69, 30, 160, 67
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION
CAPTION "Mastering Make - Info"
FONT 8, "MS Sans Serif"
BEGIN
    CTEXT           "Mastering Make: Windows Sample Program",
                    -1, 10, 10, 141, 8
    CTEXT           "Second Edition - Prentice Hall, 1994",
                    -1, 15, 26, 130, 8
    DEFPUSHBUTTON   "OK", IDOK, 56, 46, 40, 14
END
```

Makefile for Microsoft's NMAKE

```
#
# Microsoft Makefile - Windows Example
#
# Mastering Make: A Guide to Building Programs on
# DOS, OS/2, and UNIX* Systems
# Tondo, Nathanson, Yount
#
# Prentice Hall (C) 1994
#

#
# Define suffixes
#

.SUFFIXES:
.SUFFIXES: .c .obj .exe .rc .res .dlg .def

#
# Define macros
#

CC      = cl
CINC    = -I.
LINK    = link
LIBS    = slibcw
RC      = rc
RCFLAGS = -r
TARGET  = winsamp.exe

#
# Test for defined macro
#

!ifndef DEBUG
DEBUG=yes
!endif
```

```
#
# Define macros for compiling and linking
#

!if "$(DEBUG)" == "yes" || "$(DEBUG)" == "YES"
CFLAGS  = -c -W3 -G2sw -Zp -W3 -AS -Zid -Od
LFLAGS  = /co
!else
CFLAGS  = -c -W3 -G2sw -Zp -W3 -AS -O
LFLAGS  =
!endif

#
# Define inference rules
#

.c.obj:
        $(CC) $(CFLAGS) $(CINC) $*.c

.rc.res:
        $(RC) $(RCFLAGS) $*.rc

#
# Define objects used to create the target
#

OBJS    = $(TARGET:.exe=.obj)

#
# Define the target to build
#

$(TARGET): $(OBJS) $(TARGET:.exe=.def)  \
                   $(TARGET:.exe=.res)
        $(LINK) $(LFLAGS) @<<$*.lnk
$(OBJS)
$@
NUL
$(LIBS)
```

```
$(TARGET:.exe=.def)
<<nokeep
        $(RC) -t $(TARGET:.exe=.res) $@

#
# Define module definition file used to create executable
#

$(TARGET:.exe=.def):
        @echo $@ @<<$*.def
NAME          WinSamp

DESCRIPTION   'Mastering Make Example - Microsoft'

EXETYPE       WINDOWS

CODE          PRELOAD MOVEABLE DISCARDABLE
DATA          PRELOAD MOVEABLE MULTIPLE

HEAPSIZE      1024
STACKSIZE     5120
<<keep

#
# Define indirect dependencies
#

$(TARGET:.exe=.obj): $(TARGET:.exe=.c)

$(TARGET:.exe=.res): $(TARGET:.exe=.rc)  \
                     $(TARGET:.exe=.dlg)
```

Makefile for Borland's MAKE

```
#
# Borland Makefile - Windows Example
#
# Mastering Make: A Guide to Building Programs on
# DOS, OS/2, and UNIX* Systems
# Tondo, Nathanson, Yount
#
# Prentice Hall (C) 1994
#

#
# Define suffixes
#

.suffixes:
.suffixes: .c .obj .exe .rc .res .dlg .def

#
# Define other makefile directives
#

.autodepend

.path.obj = .

#
# Define macros
#

CC      = bcc
CINC    = -I.
LINK    = tlink
LIBS    = cws graphics import
RC      = rc
RCFLAGS = -r
TARGET  = winsamp.exe
```

```
#
# Test for defined macro
#

!if !$d(DEBUG)
DEBUG=yes
!endif

#
# Define macros for compiling and linking
#

!if $(DEBUG) == "yes" || $(DEBUG) == "YES"
CFLAGS  = -c -R -ms -2 -v -d -W
LFLAGS  = -v -x -c -P- -Twe
!else
CFLAGS  = -c -R -ms -2 -d -W -O
LFLAGS  = -x -c -P- -Twe
!endif

#
# Define inference rules
#

.c.obj:
        $(CC) $(CFLAGS) $(CINC) $*.c

.rc.res:
        $(RC) $(RCFLAGS) $*.rc

#
# Define objects used to create the target
#

OBJS    = $(TARGET:.exe=.obj)

WINOBJ  = c0ws.obj

#
```

```
# Define the target to build
#

$(TARGET): $(OBJS) $(TARGET:.exe=.def)  \
                   $(TARGET:.exe=.res)
        $(LINK) $(LFLAGS) @&&|
$(WINOBJ)  $(OBJS)
$@
nul
$(LIBS)
$(TARGET:.exe=.def)
|
        $(RC) -t $(TARGET:.exe=.res) $@

#
# Define module definition file used to create executable
#

$(TARGET:.exe=.def):
        @copy &&|
NAME         WinSamp

DESCRIPTION  'Mastering Make Example - Borland'

EXETYPE      WINDOWS

CODE         PRELOAD MOVEABLE DISCARDABLE
DATA         PRELOAD MOVEABLE MULTIPLE

HEAPSIZE     1024
STACKSIZE    5120
| $@

#
# Define indirect dependencies
#

$(TARGET:.exe=.obj): $(TARGET:.exe=.c)
```

```
$(TARGET:.exe=.res): $(TARGET:.exe=.rc)  \
                     $(TARGET:.exe=.dlg)
```

Appendix G

Presentation Manager Sample

The Program

```
//
//  Program Name: pmsamp.c
//
//  Description:
//    This program is an OS/2 Presentation Manager
//    sample that displays 'mastering make' on
//    the screen.
//
//  Mastering Make - Tondo, Nathanson, Yount
//  Prentice Hall (C) 1994
//


// Include headers
#include    <stdio.h>
#include    <string.h>
#include    <stdlib.h>

// Include OS/2 headers
#define     INCL_PM
#define     INCL_GPI
#include    <os2.h>

// Define symbolic constants
```

```
#define     ID_EXIT     150
#define     ID_INFODLG  200
#define     ID_INFO     210
#define     ID_MAINWND  500

// Declare function prototypes
void        main        (void);

MRESULT EXPENTRY  DisplayInfo  (HWND hwnd, ULONG msg,
                                MPARAM mp1, MPARAM mp2);
MRESULT EXPENTRY  SetWindProc  (HWND hwnd, ULONG msg,
                                MPARAM mp1, MPARAM mp2);

//  Global variables

HWND     HwndClient,            // Client window handle
         HwndFrame;             // Frame window handle

ULONG    FlCreate = FCF_TITLEBAR | FCF_SIZEBORDER |
                    FCF_SHELLPOSITION |
                    FCF_TASKLIST | FCF_MINMAX     |
                    FCF_MENU;
// main -
//  This function initializes the PM interface and
//  defines the window area that it will use. It also
//  initializes the message queue that is used by the
//  graphic engine.

void main(void)
{

   // Define local variables

   int      height = 0;
   BOOL     rc;
   HAB      hab;
   HMQ      hmq;
   QMSG     qmsg;
```

```
// Start interface

hab = WinInitialize(0);

if ( hab == 0 )
{
   return ;
}

// Start the message queue

hmq = WinCreateMsgQueue(hab, 0);

if ( hmq == 0 )
{
   WinTerminate(hab);
   return ;
}

// Register a window class

rc = WinRegisterClass(hab,
                      "MyWindow",
                      SetWindProc,
                      CS_SIZEREDRAW,
                      0);

// Check the results of the window registering

if ( rc == 0 )
{
   WinTerminate(hab);
   return ;
}

// Create a standard window

HwndFrame = WinCreateStdWindow(HWND_DESKTOP,
                               0L,
```

```
                                    &FlCreate,
                                    "MyWindow",
                                    "",
                                    0L,
                                    (HMODULE)NULL,
                                    ID_MAINWND,
                                    &HwndClient);

if ( HwndFrame == 0 )
{
   return ;
}

// Get the system values for the screen

height = WinQuerySysValue(HWND_DESKTOP, SV_CYSCREEN) -
                          (HWND_DESKTOP, SV_CYICON);

// Change the window size

WinSetWindowPos(HwndFrame,
                HWND_TOP,
                0,
                WinQuerySysValue(HWND_DESKTOP,SV_CYSCREEN)
                                              - height,
                WinQuerySysValue(HWND_DESKTOP,SV_CXSCREEN),
                                 height,
                                 SWP_SHOW     | SWP_ZORDER |
                                 SWP_ACTIVATE | SWP_SIZE   |
                                 SWP_MOVE);

// Set the window with a text

WinSetWindowText(HwndFrame, "Mastering Make PM Sample");

// Get and dispatch messages from the application
// message queue until WinGetMsg returns FALSE,
// indicating a WM_QUIT message.
```

```
    while( WinGetMsg( hab, &qmsg, (HWND)NULL, 0, 0 ) )
    {
       WinDispatchMsg( hab, &qmsg );
    }

    // Destroy the message queue and terminate the window
    // anchor block

    WinDestroyWindow (HwndFrame);
    WinDestroyMsgQueue( hmq );
    WinTerminate( hab );

    return ;

}

// DisplayInfo -
//   This function displays the info window on the screen.


MRESULT EXPENTRY DisplayInfo(HWND hwnd, ULONG msg,
                             MPARAM mp1, MPARAM mp2)
{

    // Check the message

    switch ( msg )
    {
       case WM_INITDLG:
          break;
       case WM_COMMAND:
          WinDismissDlg(hwnd, TRUE);
          break;
       default:
          return WinDefDlgProc(hwnd, msg, mp1, mp2);
          break;
    }

    // Return FALSE
```

```
   return (MRESULT) FALSE;

}

#define         MAXLINE         100

// SetWindProc -
//   This function interprets the messages from PM
//   and performs the necessary tasks.

MRESULT EXPENTRY SetWindProc(HWND hwnd, ULONG msg,
                             MPARAM mp1, MPARAM mp2)
{

   // Define local variables

   static char    line1[MAXLINE],
                  line2[MAXLINE];
   HPS            hps;
   POINTL         ptl;
   RECTL          rec;

   // Check the message

   switch( msg )
   {
      case WM_CREATE:

         // Create the data strings to be printed on
         // the screen

         strcpy(line1,
           "Mastering Make: A Guide to Build Programs on "
           "DOS and UNIX* Systems");
         strcpy(line2,
           "              and will now include WINDOWS "
           "and OS/2 Support");
         break;
```

```
case WM_COMMAND:

   // Interpret the pull-down menus

   switch ( SHORT1FROMMP( mp1 ) )
   {
      case ID_INFO:
         WinDlgBox(HWND_DESKTOP,
                   HWND_DESKTOP,
                   (PFNWP) DisplayInfo,
                   (HMODULE) 0,
                   ID_INFODLG,
                   NULL);
         break;
      case ID_EXIT:
         WinPostMsg(hwnd, WM_CLOSE, NULL, NULL);
         break;
      default:
         return WinDefWindowProc(hwnd, msg, mp1, mp2);
   }
   break;
case WM_PAINT:

   // Write information to the window presentation
   // space

   WinQueryWindowRect(hwnd, &rec);
   hps = WinBeginPaint(hwnd, (HPS)NULL, &rec );
   WinFillRect(hps, &rec, SYSCLR_WINDOW);
   GpiSetColor(hps, CLR_NEUTRAL );
   ptl.x = 100;
   ptl.y = 250;
   GpiCharStringAt(hps,
                   &ptl,
                   (LONG) strlen(line1),
                   line1);

   ptl.x = 100;
   ptl.y = 220;
```

```
            GpiCharStringAt(hps,
                            &ptl,
                            (LONG) strlen(line2),
                            line2);

            WinEndPaint( hps );
            break;
         case WM_CLOSE:

            // Post a WM_QUIT message

            WinPostMsg(hwnd, WM_QUIT, NULL, NULL);
            break;
         default:

            // Return information to interface

            return WinDefWindowProc(hwnd, msg, mp1, mp2 );
      }

      // Return FALSE

      return (MRESULT) FALSE;

   }
```

The .rc File

```
//
//  Resource Name: pmsamp.rc
//
//  Description:
//    This file contains the Presentation Manager
//    resource information.
//  Mastering Make - Tondo, Nathanson, Yount
//
//  Prentice Hall (C) 1994

#include    <os2.h>

#define     ID_EXIT     150
#define     ID_INFODLG  200
#define     ID_INFO     210
#define     ID_MAINWND  500

MENU  ID_MAINWND  PRELOAD
{
   SUBMENU  "~Info",    ID_INFO
   {
      MENUITEM "~Info", ID_INFO
   }

   SUBMENU  "~Exit",    ID_EXIT
   {
      MENUITEM "~Exit", ID_EXIT
   }
}

rcinclude   pmsamp.dlg
```

The .dlg File

```
// Mastering Make: OS/2 PM Sample Dialog
//
// Mastering Make - Tondo, Nathanson, Yount
// Prentice Hall (C) 1994
//

DLGTEMPLATE ID_INFODLG LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG  "Mastering Make Help Dialog", ID_INFODLG,
            101, 105, 182, 65, WS_VISIBLE, FCF_TITLEBAR
    BEGIN
        CTEXT           "Mastering Make: OS/2 PM Sample",
                        101, 5, 49, 169, 8
        CTEXT           "Second Edition - Prentice Hall 1994",
                        102, 3, 37, 173, 8
        DEFPUSHBUTTON   "OK", DID_OK, 64, 9, 40, 14
    END
END
```

Makefile for IBM's NMAKE

```
#
# IBM Makefile for OS/2
# Presentation Manager Example
#
# Mastering Make: A Guide to Building Programs on
# DOS, OS/2, and UNIX* Systems
# Tondo, Nathanson, Yount
#
# Prentice Hall (C) 1994
#

#
# Define suffixes
#

.SUFFIXES:
.SUFFIXES: .c .obj .exe .rc .res .dlg .def

#
# Define macros
#

CC      = icc
LINK    = link386
LIBS    = os2386 os2
RC      = rc
TARGET  = pmsamp.exe
LIBPATH = h:\ibmcpp\lib
CINC    = -I.
RCFLAGS = -r

#
# Test for defined macro
#

!ifndef DEBUG
DEBUG=yes
```

```
!endif

#
# Define macros for compiling and linking
#

!if "$(DEBUG)" == "yes" || "$(DEBUG)" == "YES"
CFLAGS  = -c -Gd- -Se -Re -ss -Ms -Ti+ -W3 -G3
LFLAGS  = /co
!else
CFLAGS  = -c -Gd- -Se -Re -ss -Ms -W3 -G3
LFLAGS  =
!endif

#
# Define inference rules
#

.c.obj:
        $(CC) $(CFLAGS) $(CINC) $*.c

.rc.res:
        $(RC) $(RCFLAGS) $*.rc

#
# Define objects used to create the target
#

OBJS = $(TARGET:.exe=.obj)

#
# Define the target to build
#

$(TARGET): $(OBJS) $(TARGET:.exe=.def) \
           $(TARGET:.exe=.res)
        $(LINK) $(LFLAGS) @<<$(TARGET:.exe=.lnk)
$(OBJS)
$@
```

```
NUL
$(LIBS)
$(TARGET:.exe=.def)
<<keep
        $(RC) $(TARGET:.exe=.res) $@

#
# Define module definition file used to create executable
#

$(TARGET:.exe=.def):
        @echo $@ @<<$@
; Mastering Make - OS/2 PM Module Definition File Example

NAME         PMSAMP      WINDOWAPI

DESCRIPTION  'Mastering Make Example - IBM'

PROTMODE
<<keep

#
# Define indirect dependencies
#

$(TARGET:.exe=.obj): $(TARGET:.exe=.c)

$(TARGET:.exe=.res): $(TARGET:.exe=.rc) \
                     $(TARGET:.exe=.dlg)
```

Makefile for Borland's MAKE

```
#
# Borland Makefile for OS/2
# Presentation Manager Example
#
# Mastering Make: A Guide to Building Programs on
# DOS, OS/2, and UNIX* Systems
# Tondo, Nathanson, Yount
#
# Prentice Hall (C) 1994
#

#
# Define suffixes
#

.suffixes:
.suffixes: .c .obj .exe .rc .res .dlg .def

#
# Define other makefile directives
#

.autodepend

.path.obj = .

#
# Define macros
#

CC      = bcc
CINC    = -I.
LINK    = tlink
LIBS    = c2 os2
BRCC    = brcc
RC      = rc
RCFLAGS = -r
```

```
TARGET  = pmsamp.exe

#
# Test for defined macro
#

!if !$d(DEBUG)
DEBUG=yes
!endif

#
# Define macros for compiling and linking
#

!if $(DEBUG) == "yes" || $(DEBUG) == "YES"
CFLAGS  = -c -3 -v
LFLAGS  = -v -Toe -aa
!else
CFLAGS  = -c -3 -O1
LFLAGS  = -Toe -aa
!endif

#
# Define inference rules
#

.c.obj:
   $(CC) $(CFLAGS) $(CINC) $*.c

.rc.res:
   $(BRCC) $(RCFLAGS) $*.rc

#
# Define objects used to create the target
#

OBJS  = $(TARGET:.exe=.obj)

BCOBJ = c02.obj
```

```
#
# Define the target to build
#

$(TARGET): $(OBJS) $(TARGET:.exe=.def) \
           $(TARGET:.exe=.res)
        $(LINK) $(LFLAGS) @&&|
$(BCOBJ)  $(OBJS)
$@
nul
$(LIBS)
$(TARGET:.exe=.def)
|
        $(RC) $(TARGET:.exe=.res) $@

#
# Define module definition file used to create executable
#

$(TARGET:.exe=.def):
        @copy &&|
; Mastering Make - OS/2 PM Module Definition File Example

NAME         PMSAMP      WINDOWAPI

DESCRIPTION  'Mastering Make Example - Borland'

PROTMODE
| $@

#
# Define indirect dependencies
#

$(TARGET:.exe=.obj): $(TARGET:.exe=.c)

$(TARGET:.exe=.res): $(TARGET:.exe=.rc) \
                     $(TARGET:.exe=.dlg)
```

Appendix H

X Windows Sample

The Program

```
/*
 *  Program Name: xmsamp.c
 *
 *  Description:
 *    This program is a UNIX X Window System
 *    (with OSF Motif Widgets) example
 *    that displays 'mastering make' on
 *    the screen.
 *
 *  Written by Thomas Bagli
 *  Mastering Make: A Guide to Building Programs on
 *  DOS, OS/2, and UNIX* Systems
 *  Tondo, Nathanson, Yount
 *
 *  Prentice Hall (C) 1994
 *
 */

/* Include system headers */
#include <stdio.h>

/* Include X intrinsic headers */
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
```

```
/* Include motif headers */
#include <Xm/Form.h>
#include <Xm/RowColumn.h>
#include <Xm/CascadeB.h>
#include <Xm/PushB.h>
#include <Xm/Label.h>
#include <Xm/MessageB.h>


/* Declare external function prototypes */
extern void exit (int);


/* Define symbolic constants */
#define MAIN_MESSAGE "\
Mastering Make: A Guide to Building Programs on DOS,\n\
                              OS/2, and UNIX* Systems"
#define MAIN_TITLE "Mastering Make UNIX X/Motif Example"

#define INFO_MESSAGE "\
Mastering Make: UNIX X Window System Motif Widgets Example\n\
                         Second Edition - Prentice Hall 1994"
#define INFO_TITLE "Mastering Make Info Dialog"


/* Global variables */

XtAppContext app_context;

XmFontList font_list;


/* Define OK callback function */
static void DoOK (w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
    Widget top = (Widget) client_data;
```

```
    XtPopdown (top);
    XtDestroyWidget (top);
}


/* Define information callback function */
static void DoInfo (w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
    Arg args[8];
    Cardinal nargs;
    Widget parent, top, info_dialog;
    XmString msg;

    parent = (Widget) client_data;

    nargs = 0;
    XtSetArg (args[nargs], XmNiconName, INFO_TITLE);
    nargs++;
    XtSetArg (args[nargs], XmNtitle, INFO_TITLE);
    top = XtCreatePopupShell ("info_dialog",
    topLevelShellWidgetClass, parent, args, nargs);

    msg = XmStringCreateLtoR (INFO_MESSAGE,
                XmSTRING_DEFAULT_CHARSET);

    nargs = 0;
    XtSetArg (args[nargs], XmNbuttonFontList, font_list);
    nargs++;
    XtSetArg (args[nargs], XmNlabelFontList, font_list);
    nargs++;
    XtSetArg (args[nargs], XmNmessageString, msg);
    nargs++;
    info_dialog = XmCreateMessageBox (top, "info_dialog",
                          args, nargs);

    XtUnmanageChild (XmMessageBoxGetChild (info_dialog,
```

```
                    XmDIALOG_CANCEL_BUTTON));
    XtUnmanageChild (XmMessageBoxGetChild (info_dialog,
                    XmDIALOG_HELP_BUTTON));

    XtManageChild (info_dialog);

    XtAddCallback (info_dialog, XmNokCallback, DoOK, top);

    XmStringFree (msg);


    XtPopup (top, XtGrabNone);
}


/* Define exit callback function */
static void DoExit (w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
    XtDestroyApplicationContext (app_context);
    exit (0);
}


Widget CreateMenus (top, form)
Widget top, form;
{
    Arg args[8];
    Cardinal nargs;
    Widget menu_bar;
    Widget info_pane, info_menu, info_button;
    Widget exit_pane, exit_menu, exit_button;

    /* Create the menubar */

    nargs = 0;
    XtSetArg (args[nargs], XmNspacing, 4);
    nargs++;
```

```
XtSetArg (args[nargs], XmNtopAttachment, XmATTACH_FORM);
nargs++;
XtSetArg (args[nargs], XmNleftAttachment, XmATTACH_FORM);
nargs++;
XtSetArg (args[nargs], XmNrightAttachment, XmATTACH_FORM);
nargs++;
menu_bar = XmCreateMenuBar (form, "menu_bar",
                            args, nargs);
XtManageChild (menu_bar);


/* Create info pane, cascade and push button menu */

nargs = 0;
info_pane = XmCreatePulldownMenu (menu_bar, "info_pane",
                                  args, nargs);

nargs = 0;
XtSetArg (args[nargs], XmNfontList, font_list);
nargs++;
XtSetArg (args[nargs], XmNsubMenuId, info_pane);
nargs++;
XtSetArg (args[nargs], XmNmnemonic, 'H');
nargs++;
info_menu = XmCreateCascadeButton (menu_bar, "Info",
                                   args, nargs);
XtManageChild (info_menu);

nargs = 0;
XtSetArg (args[nargs], XmNfontList, font_list);
nargs++;
info_button = XmCreatePushButton (info_pane, "Info",
                                  args, nargs);
XtAddCallback (info_button, XmNactivateCallback,
               DoInfo, top);
XtManageChild (info_button);


/* Create exit pane, cascade and push button menu */
```

```
    nargs = 0;
    exit_pane = XmCreatePulldownMenu (menu_bar,
                                      "exit_pane",
                                      args, nargs);

    nargs = 0;
    XtSetArg (args[nargs], XmNfontList, font_list);
    nargs++;
    XtSetArg (args[nargs], XmNsubMenuId, exit_pane);
    nargs++;
    XtSetArg (args[nargs], XmNmnemonic, 'E');
    nargs++;
    exit_menu = XmCreateCascadeButton (menu_bar, "Exit",
                                       args, nargs);
    XtManageChild (exit_menu);

    nargs = 0;
    XtSetArg (args[nargs], XmNfontList, font_list);
    nargs++;
    exit_button = XmCreatePushButton (exit_pane,
                            "Exit", args, nargs);
    XtAddCallback (exit_button, XmNactivateCallback,
                   DoExit, NULL);
    XtManageChild (exit_button);

    return (menu_bar);
}


void main (argc, argv)
int argc;
char **argv;
{

    /* Define local variables */

    char *font_str = "*-Adobe-Times-Bold-R-Normal--14-*";
```

```
Arg args[12];
Cardinal nargs;
Widget top, form, menu_bar, message;
XFontStruct *font;
XmString msg;

nargs = 0;
XtSetArg (args[nargs], XmNminWidth, 500);
nargs++;
XtSetArg (args[nargs], XmNminHeight, 300);
nargs++;
XtSetArg (args[nargs], XmNtitle, MAIN_TITLE);
nargs++;
top = XtAppInitialize (&app_context, "XMsamp", NULL, 0,
                       &argc, argv, NULL, args, nargs);

/* Load the specified font */

font = XLoadQueryFont (XtDisplay (top), font_str);
if (font == NULL)
    fprintf (stderr, "Could not load font: \"%s\"\n",
             font_str);
font_list = XmFontListCreate (font,
             XmSTRING_DEFAULT_CHARSET);

nargs = 0;
XtSetArg (args[nargs], XmNfontList, font_list);
nargs++;
XtSetValues (top, args, nargs);


/* Create the outer form */
nargs = 0;
form = XmCreateForm (top, "form", args, nargs);
XtManageChild (form);


/* Create the menus */
```

```
menu_bar = CreateMenus (top, form);


/* Create message string and message box */

msg = XmStringCreateLtoR (MAIN_MESSAGE,
                          XmSTRING_DEFAULT_CHARSET);

nargs = 0;
XtSetArg (args[nargs], XmNlabelString, msg);
nargs++;
XtSetArg (args[nargs], XmNfontList, font_list);
nargs++;
XtSetArg (args[nargs], XmNmarginTop, 150);
nargs++;
XtSetArg (args[nargs], XmNmarginLeft, 150);
nargs++;
XtSetArg (args[nargs], XmNmarginRight, 150);
nargs++;
XtSetArg (args[nargs], XmNmarginBottom, 150);
nargs++;
XtSetArg (args[nargs], XmNtopWidget, menu_bar);
nargs++;
XtSetArg (args[nargs], XmNtopAttachment,
          XmATTACH_WIDGET);
nargs++;
XtSetArg (args[nargs], XmNleftAttachment,
          XmATTACH_FORM);
nargs++;
XtSetArg (args[nargs], XmNrightAttachment,
          XmATTACH_FORM);
nargs++;
XtSetArg (args[nargs], XmNbottomAttachment,
          XmATTACH_FORM);
nargs++;
message = XmCreateLabel (form, "message",
                         args, nargs);
XtManageChild (message);
```

```
        XmStringFree (msg);

        XtRealizeWidget (top);

        XtAppMainLoop (app_context);
    }
```

The Makefile for UNIX's MAKE

```
#
# Makefile for UNIX X Window System
# (with Motif Widgets) Example
#
# Mastering Make: A Guide to Building Programs on
# DOS, OS/2, and UNIX* Systems
# Tondo, Nathanson, Yount
#
# Prentice Hall (C) 1994
#

RM              = rm -f

CFLAGS          = -g  -O

EXTENSIONLIB    = -lXext
XLIB            = $(EXTENSIONLIB) -lX11
XMLIB           = -lXm
XMULIB          = -lXmu
XTOOLLIB        = -lXt

LOCAL_LIBRARIES = $(XMLIB) $(XMULIB) $(XTOOLLIB) $(XLIB)

OBJS            = xmsamp.o

xmsamp: $(OBJS)
                $(RM) $@
                $(CC) -o $@ $(OBJS) $(LOCAL_LIBRARIES)
```

Appendix I

Depends Utility

Purpose

Generate dependency list for the `.obj` files that appear in a DOS or OS/2 makefile. This is a simple dependency generator meant to provide an example of how to generate dependencies from a makefile. This utility does not take into consideration all the features presented in this book.

Syntax

```
depends  -IPath [ -IPath ... ]
                [ -e ]
                [ -ffilename ]
                [ -ofilename ]
                [ -Q ]

depends  -I Path [ -I Path ... ]
                 [ -e ]
                 [ -f filename ]
                 [ -o filename ]
                 [ -Q ]
```

Description

The `depends` program reads a makefile and collects the `.obj` names in a dependency list. It then reads the source file for the corresponding `.obj` file in the dependency list looking for `include` files. `depends` handles `.c`

and `.asm` source files. For each `include` file, `depends` searches the path(s) indicated with the `I` option(s) for an occurrence of the `include` file and produces a dependency line for the makefile.

The output from the program goes to `stdout` (the standard output in DOS and OS/2 is the screen) or to a file that was selected by the user with the `o` option.

Here are the options that the program supports:

- `-e`
  Forces the program to ignore the environment variable `INCLUDE`.
- `-f` *filename* or `-f`*filename*
  The name of the makefile to be parsed. The default name is `makefile.old`.
- `-I` *includepath* or `-I`*includepath*
  The path where `depends` searches for `include` files. There is *no* default path so you must specify at least one `I`; the maximum number of `I`s is 10.
- `-o` *filename* or `-o`*filename*
  The name of the output file where the results will be placed.
- `-Q`
  Converts all relative paths to fully qualified paths.

You can use a dash (`-`) in front of an option, as we show here, or a slash (`/`) if you prefer the PC style for parameters. The order of options is not important.

Examples

```
depends  -I. -I\c\include -o makefile
```

`depends` will search the current directory (`.`) and `\c\include` for the `include` files. The input file is the default `makefile.old`. The output will be written to the file `makefile`.

Notice that the `-I` and `-f` options can have their arguments next to them or separated by blanks or tabs. For example,

```
depends  -I. -I\c\include -omakefile -e
```

and

```
depends  -I . -I \c\include -o makefile -e
```

are equivalent.

In the next example,

```
depends  -I. -I..\c\include -Q -f temp.mak -o makefile
```

`depends` will read the makefile `temp.mak`; it will then search `.` and `..\c\include` directories; and it will produce fully qualified dependency paths because of `-Q`. The output will be written to `makefile`.

```
depends -f temp.mak -I. -Q -o makefile
```

`depends` will read the makefile `temp.mak`; it will then search `.` and the directories listed in the environment variable `INCLUDE`; and it will produce fully qualified dependency paths because of `-Q`. The output will be written to `makefile`. Note that when we use NMAKE and invoke `depends` from the makefile, we need to add an extra pair of `%`s around the environment variable `INCLUDE` because `%` is a special character in NMAKE. The preceding example becomes:

```
depends  -f temp.mak -I. -Q -o makefile
```

Note: It is important that you do *not* send the output to the file that is also the input to `depends`. For example, this is an error:

```
depends  -I.  -f tmpfile  -o tmpfile
```

DOS and OS/2 will open `tmpfile` for output and will truncate the size of an existing file to zero before `depends` has the chance to read `tmpfile`.

Observations

More sophisticated dependency tools may be available on your system. Some versions of *make*, such as Borland's MAKE, have dependency analysis built in. Consult your local guru.

Outline for the depends Utility

```
handle command line arguments

Scan() - scan the input file for .obj
    Dependencies() - find xxx.obj names
        AddRecord() - add xxx.obj to the list of
                      .obj names

ScanFile() - for each xxx.obj in the list
    is there a xxx.c file?
        ScanC()
            search for '#include'
            search the -I directories for the file name
                following the '#include'
            output a dependency line for the file name
    or is there a xxx.asm file?
        ScanAsm()
            search for 'include'
            search the -I directories for the file name
                following the 'include'
            output a dependency line for the file name
```

The Program

```
//
//  Program Name: depends
//
//  Description:
//   This program creates a set of dependencies for
//   assembler and C programs.
//
//   We have modified the program to allow it to
//   work within a DOS or OS/2 environment.
//
//   The input is a valid makefile. The output is the
//   original file plus the list of dependencies.
//   The output can be written to either an existing
//   file or a new file (if specified) or the default
//   will be standard out.
//
//   Operating System: DOS or OS/2
//
//   Mastering Make: A Guide to Building Programs on
//                   DOS, OS/2, and UNIX* Systems
//   Prentice Hall (C) 1994

//  Include Files

#include      <stdio.h>
#include      <string.h>
#include      <stdlib.h>
#include      <ctype.h>
#include      <malloc.h>
#include      <direct.h>

#ifdef DOS
#include      <dos.h>
#else

//  Include OS/2 Specific Include Files
```

```
#define         INCL_DOSFILEMGR
#define         INCL_NOPM
#include        <os2.h>

#endif

//  Symbolic Constants

#define          FAIL                    1
#define          MAXFILENAME             15
#define          MAXINCPATHS             20
#define          MAXLINE                 256
#define          NO                      0
#define          STANDARDOUT             0
#define          WRITEFILE               2
#define          YES                     1

#ifdef DOS
#define          FIND                    _dos_findfirst
#endif

//  Function Prototypes

int      main          (int argc, char *argv[]);
void     AddRecord     (char *src);
void     Dependencies  (char *line);
void     Error         (char *message);
int      GetFile       (int argc, char *argv[], int cnt);
int      GetOutput     (int argc, char *argv[], int cnt);
int      GetPath       (int argc, char *argv[], int cnt);
void     *GetMemory    (unsigned nbytes);
void     IncPathsCnt   (int *IncCnt);
int      LoadInclude   (void);
void     OpenFile      (char *filename);
void     Scan          (FILE *fileptr);
void     ScanAsm       (FILE *fileptr, char *filename);
void     ScanC         (FILE *fileptr, char *filename);
void     ScanFile      (void);
void     SearchPaths   (char *iname);
```

```
void    Usage         (void);

//  Global Variables

char    FileName[MAXFILENAME],
        *IncPaths[MAXINCPATHS],
        NewFilename[MAXFILENAME],
        OutputFile[MAXFILENAME];

int     EnvFlag = 1,        // Set Use 'Include' from
                            // environment
        IncCnt  = -1,       // Total number of include paths
        NeedContinuation = NO,
        OutputOption = 0, // Output option  (default stdout)
        Qflags  = NO;       // Do not use fully qualified paths

FILE    *OutFile;

struct target
{
        char            *name;
        struct target   *next;
};

struct target *TargetsBegin, *TargetsEnd;

// main: main function for 'depends'
int main (int argc, char *argv[])
{
        char    line[MAXLINE];

        int     cnt;        // Argument count

        FILE    *fileptr;   // Pointer to input file

                            // Initialize default file name
        strcpy(FileName, "makefile.old");
```

```
if (argc < 2)        // Not enough arguments?
    Usage();
else                 // Check command line options
{
    for (cnt = 1; cnt < argc; cnt++)
    {
        if ( argv[cnt][0] != '-' &&
             argv[cnt][0] != '/')
        {
            sprintf(line,
                    "Invalid Option ==> [ %s ]\n",
                    argv[cnt]);
            Error(line);
            Usage();
        }
        else
        {
            switch(argv[cnt][1])
            {
                case 'e':
                case 'E':
                // Set to exclude environment variable
                // 'INCLUDE'
                    EnvFlag = 0;
                    break;
                case 'i':
                case 'I':
                // Get include path information
                    cnt = GetPath(argc, argv, cnt);
                    break;
                case 'f':
                case 'F':
                // Get makefile name
                    cnt = GetFile(argc, argv, cnt);
                    break;
                case 'o':
                case 'O':
                // Get output file name
                    cnt = GetOutput(argc, argv, cnt);
```

```
                    break;
                case 'q':
                case 'Q':
                // Use fully qualified paths
                    Qflags = YES;
                    break;
                default:
                    sprintf(line,
                            "Invalid Option "
                            "==> [ %s ]\n",
                            argv[cnt]);
                    Error(line);
                    Usage();
                    break;
            }
        }
    }
}

if (EnvFlag == 1)       // Check environment flag
    LoadInclude();      // Load environment 'include'
else
{
    if (IncCnt < 1)     // Need at least one -I
        Usage();
}
                        // Open input file
if ( (fileptr = fopen(FileName, "r")) == NULL )
{
    sprintf(line,
            "Can't open file ==> [ %s ]\n", FileName);
    Error(line);
    exit(FAIL);
}
Scan(fileptr);          // Scan the input file
fclose(fileptr);

ScanFile();             // Scan each source file
return 0;
```

```
}

// AddRecord: add a new record (with file name) to list
void AddRecord(char *filename)
{
    register struct target *ptr;

    // Filename in the linked list?
    for (ptr = TargetsBegin; ptr; ptr = ptr->next)
    {
        if ( strcmp(ptr->name, filename ) == 0)
            return;     // Filename already in the list
    }
    // Get memory space for new node
    ptr = GetMemory(sizeof(struct target));

    // Add node to the linked list
    if (!TargetsEnd)    // The list is empty
        TargetsBegin = ptr;
    else                // Add node to end of the list
        TargetsEnd->next = ptr;
    ptr->next  = NULL;
    ptr->name  = filename;
    TargetsEnd = ptr;
}

// Dependencies: find .obj file name and add it to the list
void  Dependencies(char *line)
{
    register char   *src = strdup(line);
    register char   *ptr;

    for (ptr = strstr(src, ".obj"); ptr;
         ptr = strstr(ptr, ".obj"))
    {
        // Mark the end of the file name
        *(ptr + 4) = '\0';
        // Back up to beginning of the file name
        for ( ; ptr >= src && !isspace(*ptr); ptr--)
```

```
            ;
        // add the file name to linked list of file names
        AddRecord(++ptr);

        // point past current file name
        ptr += strlen(ptr) + 1;
    }
}

// Error: display error message
void Error(char *message)
{
    fprintf(stderr, "%s", message);
}

// GetFile: get file name from the command line
int GetFile(int argc, char *argv[], int cnt)
{
    char    line[MAXLINE];  // Local buffer

                            // -ffilename
    if (argv[cnt][2] != '\0')
        strcpy(FileName, &argv[cnt][2]);
    else
    {                       // -f  filename
        if (cnt+1 >= argc)
        {
            sprintf(line, "Missing Filename ==> [ %s ]\n",
                    argv[cnt]);
            Error(line);
            Usage();
        }
        strcpy(FileName, argv[++cnt]);
    }
    return cnt;
}

// Get output file name
int GetOutput(int argc, char *argv[], int cnt)
```

```
{

    char    line[MAXLINE];      // Local buffer

    if (cnt+1 > argc)
    {
        strcpy(line, "Missing File Name\n");
        Error(line);
        Usage();
    }

    OutputOption = WRITEFILE;
    OpenFile(argv[++cnt]);

    return cnt;
}


// GetPath: get include path and environment variables
int GetPath(int argc, char *argv[], int cnt)
{
    char    line[MAXLINE],  // Local buffer
            *endptr,
            *environ,       // Pointer to environment var
            *ptr;           // Pointer to include path

    if (argv[cnt][2] != '\0')
                            // -iincludepath
        ptr = &argv[cnt][2];
    else
    {                       // -i  includepath
        if (cnt+1 >= argc)
        {
            sprintf(line, "Missing Path ==> [ %s ]\n",
                    argv[cnt]);
            Error(line);
            Usage();
        }
        ptr = &argv[++cnt][0];
```

```
    }

    if (*ptr != '%')        // Include path
    {
        IncPathsCnt(&IncCnt);
        IncPaths[IncCnt] = GetMemory(strlen(ptr)+1);
        strcpy(IncPaths[IncCnt], ptr);
    }
    else                    // Environment variable
    {
        if ((endptr = strstr(++ptr, "%")) == NULL)
        {
            sprintf(line, "Invalid Option ==> [ %s ]\n",
                    argv[cnt]);
            Error(line);
            Usage();
        }
        else                // Valid environment var
        {
            *endptr = '\0'; // Terminate environment var
            strupr(ptr);
            if ((environ = getenv(ptr)) == NULL)
            {
                sprintf(line,
                        "Unknown environment variable "
                        "==> %s\n", environ);
                Error(line);
                exit(FAIL);
            }
            else            // Parse environment
            {
                strcpy(line, environ);
                for (ptr = strtok(line, ";"); ptr;
                     ptr = strtok(NULL, ";"))
                {
                    IncPathsCnt(&IncCnt);
                    IncPaths[IncCnt] =
                    GetMemory(strlen(ptr)+1);
                    strcpy(IncPaths[IncCnt], ptr);
```

```
                }
            }
        }
    }
    return cnt;     // Index of the last argument parsed
}

// GetMemory: get nbytes of memory
void *GetMemory(unsigned nbytes)
{
    char    *ptr = malloc(nbytes);

    if (!ptr)
    {
        Error("GetMemory: cannot allocate space\n");
        exit(FAIL);
    }
    return ptr ;
}

// IncPathsCnt: increment IncCnt and test upper limit
void IncPathsCnt(int *IncCnt)
{
    if (++*IncCnt >= MAXINCPATHS)
    {
        Error("Too many include files...\n");
        Error("Program terminated.\n");
        exit(FAIL);
    }
}

// Load the environment variable INCLUDE
int LoadInclude(void)
{

    char    line[MAXLINE],
            *ptr;

    ptr = getenv("INCLUDE");    // Get environment variable
```

```
    if ( ptr == '\0' )
        return -1;              // Return no entries

    strcpy(line, ptr);

    for ( ptr = strtok(line, ";");
          ptr;                  // Start parsing line
          ptr = strtok(NULL, ";\0") )
    {
        IncPathsCnt(&IncCnt);   // Check limits
                                // Get memory
        IncPaths[IncCnt] = GetMemory(strlen(ptr)+1);
                                // Copy name
        strcpy(IncPaths[IncCnt], ptr);
    }

    return 0;

}

// Open the file for output
void OpenFile(char *filename)
{

    if ((OutFile = fopen(filename, "w")) == NULL)
    {
        Error("Can't open output file\n");
        exit(FAIL);
    }
}

// Scan: scan the input file for .obj discarding
//       .obj: and its continuation lines
void Scan(FILE *fileptr)
{
    char    line[MAXLINE];
    char    *ptr;
```

```
    while (fgets(line, MAXLINE, fileptr))
    {
        // Check for .obj in the line
        if (strstr(line, ".obj"))
        {
            if (strstr(line, ".c.obj")      ||
                strstr(line, ".asm.obj")    ||
                strstr(line, ".SUFFIXES"))
                ;       // Ignore suffixes and suffix rules
            else if (!strstr(line, ".obj:"))
                        // Collect .obj dependencies
                Dependencies(line);
            else        // It is .obj:; skip target
            {           // and continuation lines
                while ((ptr=strrchr(line, '\\')) != NULL)
                {
                    if (*(ptr+1) != '\n')
                        break;  // end of continuation
                    else if (!fgets(line, MAXLINE, fileptr))
                        break;  // end of file
                }
                // discard .obj: and continuation lines
                continue;
            }
        }
        if (OutputOption != STANDARDOUT)
            fputs(line, OutFile);
        else
            fputs(line, stdout);
    }

    if (OutputOption != STANDARDOUT)
        fputs("\n", OutFile);
    else
        fputs("\n", stdout);
}

// ScanAsm: scan a .asm file for 'include' lines
void ScanAsm(FILE *fileptr, char *filename)
```

```
{
    char    line[MAXLINE],
            *ptr;

    // Print out file name
    if (OutputOption != STANDARDOUT)
        fprintf(OutFile, "%s", filename);
    else
        fprintf(stdout, "%s", filename);

    NeedContinuation = YES;

    while (fgets(line, MAXLINE, fileptr))
    {
        if (strncmp(line, "include", 7) == 0)
        {
            ptr = strtok(line, "\t ");
            ptr = strtok(NULL, "\t ");
            SearchPaths(ptr);
        }
    }
    if (OutputOption != STANDARDOUT )
        fprintf(OutFile, "\n");
    else
        fprintf(stdout, "\n");

    NeedContinuation = NO;
}

// ScanC: scan a .c file for '#include' lines
void ScanC(FILE *fileptr, char *filename)
{
    char    line[MAXLINE],
            *ptr;

    // Print out file name and a continuation line
    if ( OutputOption != STANDARDOUT)
        fprintf(OutFile, "%s", filename);
    else
```

```
            fprintf(stdout, "%s", filename);

        NeedContinuation = YES;

        while (fgets(line, MAXLINE, fileptr))
        {
            if (strncmp(line, "#include", 8) == 0)
            {
                ptr = strtok(line, "\t\"< ");
                ptr = strtok(NULL, "\t\"<> ");
                SearchPaths(ptr);
            }
        }

        if (OutputOption != STANDARDOUT)
            fprintf(OutFile, "\n");
        else
            fprintf(stdout, "\n");

        NeedContinuation = NO;
    }

    // ScanFile: scan linked list for names of target files
    void ScanFile(void)
    {
        register struct target  *ptr;
        register char           *q;
        FILE                    *fileptr;
        char filename[MAXFILENAME];

        for (ptr = TargetsBegin; ptr; ptr = ptr->next)
        {
            // Print target name; e.g.:  a.obj:

            if (OutputOption != STANDARDOUT)
                fprintf(OutFile, "%s: ", ptr->name);
            else
                fprintf(stdout, "%s: ", ptr->name);
```

```
            q  = strchr(ptr->name, '.');
            *q = '\0';
            sprintf(filename, "%s.c", ptr->name);

            if ((fileptr = fopen(filename, "r")) != NULL)
            {
                ScanC(fileptr, filename);
                fclose(fileptr);
            }
            else
            {
                sprintf(filename, "%s.asm", ptr->name);
                if ((fileptr = fopen(filename, "r")) == NULL)
                    fprintf(stderr, "Ignoring %s. . .\n",
                            ptr->name);
                else
                {
                    ScanAsm(fileptr, filename);
                    fclose(fileptr);
                }
            }
        }
    }

    // SearchPaths: search for include file and return its path
    void SearchPaths(char *iname )
    {
        char    line[MAXLINE],
                curpath[MAXLINE],
                newpath[MAXLINE];
        int     cnt;
    #ifdef DOS
        struct find_t   directory;
    #else
        HDIR            dirHandle;

        FILEFINDBUF3    directory;

        ULONG           findCount,
```

```
                    fileInfoLevel;
#endif

    for (cnt = 0; cnt <= IncCnt; cnt++)
    {
        sprintf(line, "%s\\%s", IncPaths[cnt], iname);
#ifdef DOS
        if (FIND(line, _A_NORMAL, &directory) == 0)
#else
            dirHandle = 0x0001;
            findCount = fileInfoLevel = 1;

            if (DosFindFirst(line,
                             &dirHandle,
                             FILE_NORMAL,
                             &directory,
                             sizeof(directory),
                             &findCount,
                             fileInfoLevel) == 0 )
#endif
            {
                if (Qflags == NO)   // Do not use qualified
                                    // path
                {
                    if (NeedContinuation)
                    {
                        if ( OutputOption != STANDARDOUT)
                        {
                            fprintf(OutFile, "\t\\\n");
                            fprintf(OutFile, "\t\t%s\\%-15s",
                                    IncPaths[cnt], iname);
                        }
                        else
                        {
                            fprintf(stdout, "\t\\\n");
                            fprintf(stdout, "\t\t%s\\%-15s",
                                    IncPaths[cnt], iname);
                        }
                    }
```

```
                }
                else                    // Use fully qualified
                                        // path
                {                       // Get current directory
                    getcwd(curpath, MAXLINE);
                                        // cd to include directory
                    chdir(IncPaths[cnt]);
                                        // Get qualified path
                    getcwd(newpath, MAXLINE);
                                        // Return to working dir.
                    chdir(curpath);

                    if (NeedContinuation)
                        if ( OutputOption != STANDARDOUT )
                        {
                            fprintf(OutFile, "\t\\\n");
                            fprintf(OutFile, "\t\t%s\\%-15s",
                                    newpath, iname);
                        }
                        else
                        {
                            fprintf(stdout, "\t\\\n");
                            fprintf(stdout, "\t\t%s\\%-15s",
                                    newpath, iname);
                        }
                }
                break;                  // Found include directory
            }
#ifdef OS2
            DosFindClose(dirHandle);
#endif

    }

}

// Usage: display usage message for 'depends'
```

```
void Usage(void)
{
    fprintf(stderr,
        "Usage: depends [options]\n"
        "       where each option can be any of the \n"
        "       following (in any order):\n"
        "          -e for program to not use the \n"
        "              environment variable 'INCLUDE'\n"
        "          -f filename\n"
        "              name of the makefile to parse\n"
        "              (instead of 'makefile.old')\n"
        "          -I includepath\n"
        "              Include path information\n"
        "              Minimum: one -I includepath\n"
        "              Maximum: 10  -I includepath\n"
        "          -o filename\n"
        "              name of the file to write the\n"
        "              output to.\n"
        "          -Q  Produce fully qualified paths\n");
    exit(FAIL);
}
```

Makefile for depends Using Microsoft's or IBM's NMAKE

```
#
# This is the makefile for the depends.exe. This makefile
# will create the depends.exe depending on which operating
# system you selected. This makefile will use Microsoft's or
# IBM's NMAKE command.
#
# Mastering Make: A Guide to Building Programs on DOS, OS/2,
#                 and UNIX* Systems.
#
# Tondo, Nathanson, Yount
# Prentice Hall (C) 1994
#

#
# Define suffixes
#

.SUFFIXES:
.SUFFIXES: .c .obj .exe .def


#
# Define macro
#

TARGET = depends.exe


#
# Set DEBUG as the default mode
#

!ifndef DEBUG
DEBUG=yes
!endif
```

```
#
# Set default operating system to DOS
#

!ifndef OS
OS=DOS
!endif

#
# Determine the options that need to be set
#

!if "$(OS)" == "DOS" || "$(OS)" == "dos"

#
# Define default macros
#

CC       =  cl
LINK     =  link
LIBS     =  llibc

#
# Check for debug state
#

!  if "$(DEBUG)" == "yes" || "$(DEBUG)" == "YES"
CFLAGS   =  -W3 -AL -Zi -Od -DDOS
LFLAGS   =  /co

!  else

#
# Set for non-debug state
#

CFLAGS   =  -W3 -AL -O -DDOS
LFLAGS   =
```

```
!  endif

!else        # Beginning of OS/2

#
# Define default macros
#

CC       =  icc
LINK     =  link386
LIBS     =  os2386 os2

#
# Check for debug state
#

!  if "$(DEBUG)" == "yes" || "$(DEBUG)" == "YES"
CFLAGS   =  -W3 -G3 -Ss+ -J+ -Ti+ -DOS2
LFLAGS   =  /co

!  else

#
# Set for non-debug state
#

CFLAGS   =  -W3 -G3 -SS+ -J+ -DOS2
LFLAGS   =

!   endif
!endif


!if "$(OS)" == "DOS" || "$(OS)" == "dos"

#
# Build the target (DOS)
#
```

```
$(TARGET):  depends.obj
   $(LINK)  @<<
$(TARGET:.exe=.obj)
$@
nul
$(LIBS) $(LFLAGS)
<<nokeep

!else

#
# Build the target (OS/2)
#

$(TARGET): $(TARGET:.exe=.obj) $(TARGET:.exe=.def);
   $(LINK)  @<<
$(TARGET:.exe=.obj)
$@
nul
$(LIBS) $(LFLAGS)
$(TARGET:.exe=.def)
<<nokeep

$(TARGET:.exe=.def):
   @echo $@ @<<$@
; This is the module definition file for
; the DEPENDS.EXE for OS/2.

NAME     DEPENDS.EXE       WINDOWCOMPAT

DESCRIPTION    'Depends for OS/2'

PROTMODE
<<keep

!endif

$(TARGET:.exe=.obj): $(TARGET:.exe=.c)
   $(CC)  -c  $(CFLAGS)  $*.c
```

Makefile for depends Using Borland's MAKE

```
#
# This is the makefile for the depends.exe. This makefile
# will create the depends.exe depending on which operating
# system you selected. This makefile will use Borland's
# MAKE command.
#
# Mastering Make: A Guide to Building Programs on DOS, OS/2,
#                 and UNIX* Systems.
#
# Tondo, Nathanson, Yount
# Prentice Hall (C) 1994
#

#
# Define suffixes
#

.suffixes:
.suffixes: .c .obj .exe .def


#
# Define macro
#

CC        =  bcc
LINK      =  tlink
TARGET    =  depends.exe


#
# Set DEBUG as the default mode
#

!if !$d(DEBUG)
DEBUG=yes
!endif
```

```
#
# Set default operating system to DOS
#

!if !$d(OS)
OS=DOS
!endif


#
# Determine the options that need to be set
#

!if "$(OS)" == "DOS" || "$(OS)" == "dos"

#
# Define default macros
#

LIBS     =  cl
BCOBJ    =  c:\borlandc\lib\c0l.obj


#
# Check for debug state
#

!  if "$(DEBUG)" == "yes" || "$(DEBUG)" == "YES"
CFLAGS   =  -w -ml -O- -DDOS -v
LFLAGS   =  /Tde /v

!  else

#
# Set for non-debug state
#
```

```
CFLAGS   =  -w -ml -O1 -DDOS
LFLAGS   =  /Tde

!  endif

!else        # Beginning of OS/2

#
# Define default macros
#

LIBS     =  c2  os2
BCOBJ    =  c:\bcos2\lib\c02.obj

#
# Check for debug state
#

!  if "$(DEBUG)" == "yes" || "$(DEBUG)" == "YES"
CFLAGS   =  -c -3 -w3 -O- -DOS2 -v
LFLAGS   =  /Toe /v

!  else

#
# Set for non-debug state
#

CFLAGS   =  -3 -w3 -O2 -DOS2
LFLAGS   =  /Toe

!   endif
!endif


!if "$(OS)" == "DOS" || "$(OS)" == "dos"

#
# Build the target (DOS)
```

```
#

$(TARGET):  depends.obj
   $(LINK)  @&&|
$(BCOBJ)  $(TARGET:.exe=.obj)
$@
nul
$(LIBS) $(LFLAGS)
|

!else

#
# Build the target (OS/2)
#

$(TARGET): $(TARGET:.exe=.obj) $(TARGET:.exe=.def)
   $(LINK)  @&&|
$(BCOBJ)  $(TARGET:.exe=.obj)
$@
nul
$(LIBS) $(LFLAGS)
$(TARGET:.exe=.def)
|

$(TARGET:.exe=.def):
   @copy &&|
; This is the module definition file for
; the DEPENDS.EXE for OS/2.

NAME     DEPENDS.EXE       WINDOWCOMPAT

DESCRIPTION    'Depends for OS/2'

PROTMODE
| $@

!endif
```

```
$(TARGET:.exe=.obj): $(TARGET:.exe=.c)
   $(CC)  -c  $(CFLAGS)  $*.c
```

Index

! and Borland's MAKE 99, 125, 161
! and NMAKE 69, 125, 133
!, negation expression 77, 140
! NOT operator 104, 126, 166, 187
!, preprocessor directive 69, 77, 125, 133, 140
and continuation lines 3
comment indicator 3, 8
#, example of 37
$& predefined macro 41, 50
$( ) 15, 20, 74, 104, 137, 166
$* predefined macro 24, 25, 32, 41, 198
$< predefined macro 30, 41, 198, 201
$? predefined macro 199
$@ macro 89, 116
$@ predefined macro 24, 25, 33, 41, 198
$d predefined macro 101, 102, 126, 163, 164, 187
( ) and macro names 26
- option indicator 12, 68, 98, 132, 160, 194
.asm.obj inference rule 33
.autodepend 110, 127, 172
.c.o inference rule 35, 200
.c.o: inference rule 224
.c.obj inference rule 30, 34, 35
.c: inference rule 201, 224
.com file 201
.c~.o: inference rule 203, 204, 224
.exe file 7, 201
.h files 80, 108, 143, 170
.IGNORE: pseudotarget 28, 42, 87, 150
.ignore: pseudotarget 114, 176
.inc files 80, 108, 143, 170
.o files in UNIX 35, 200
.obj files 7, 200
.s.o inference rule 201
.SILENT: pseudotarget 28, 42
.SUFFIXES: pseudotarget 28, 42, 62, 200
.suffixes: pseudotarget 28, 50, 51, 56, 62, 108, 123, 171, 185
.s~.o: inference rule 204, 224
/ option indicator 12, 68, 98, 132, 160
// comments in C 273
~ in extensions 200
: on a target line 4
? option 68, 98, 132, 160, 211, 215, 219
@ 89, 116
@copy 116
@echo 88
@echo off 88, 116

\ and ^ 79, 126, 142
\ continuation character 3, 8, 78, 107, 126, 141, 169, 195
\, example of 37
\\ in Borland's 107, 127, 169
\n 79, 127
^ and \ 79, 126, 142
^ caret 79, 126, 142

a option 68, 98, 109, 110, 132, 160, 171, 211, 215, 219
adding extension to macro 58
admin command (SCCS) 202
advantages of macros 20
AFLAGS macro 33
AS macro 33
autoresponse file 85, 112, 174

B option 98, 160, 219
b option 68, 211
backslash (\) 3, 78, 126, 141
bcc command 7, 14, 24
Borland Resource Compiler (BRCC) 180
Borland's \\ 107, 127, 169
Borland's bcc 7, 14
Borland's C compiler 7, 14
Borland's f option 61
Borland's MAKE 24, 34, 219
Borland's MAKE and ! 99, 125, 161
Borland's MAKE on DOS 97
Borland's MAKE on OS/2 159
Borland's MAKE options 98, 160, 219
Borland's MAKE, precedence of macros in 118, 129, 179, 190
BRCC 180
built-in macro $& 34, 41, 43, 50
built-in macro $* 24, 25, 32
built-in macro $< 30
built-in macro $@ 24, 25, 33
built-in macros 22
builtins.mak file 99, 116, 128, 161, 178, 220

C compiler (bcc) 7
C compiler (cc) 16
C compiler (cl) 7
C compiler (icc) 24, 34, 41, 147
C compiler, Borland's 7
C compiler, IBM's 24, 34, 41, 147
C compiler, Microsoft's 7
C compiler, UNIX system's 16
C files in UNIX 35
c option 68, 132, 201, 211, 215
caret ^ 79, 126, 142
cc command 16, 24, 41
CC macro 15, 16, 41
CC predefined macro 23, 24, 25, 41
CFLAGS empty 35
CFLAGS macro 15, 16
changing directories 46, 48, 195
cl command 7, 14, 23, 41
cmd.exe 52, 138, 187
code, return 28
command, admin (SCCS) 202
command, delta (SCCS) 205
command, get (SCCS) 203
command, implicit 30, 42

command line indentation 3, 8
command line macro 49, 50
`command.com` 52, 53, 75, 126
commands and dependency 19
commands and spaces 194
commands and tabs 194
commands and targets 5, 13
commands, explicit 5
commands, hard-coded 13, 14, 19
commands with macros 14
comment in C and // 273
comment indicator # 3, 8
common errors 61
continuation character \ 3, 8, 78, 107, 126, 141, 169, 195
continuation lines 3, 78, 107, 126, 141, 169
continuation lines and # 3
creating an environment variable 53
current target macro 24, 25
current target, synonym for 24
current target without an extension 25

`D` option 98, 102, 118, 160, 164, 179, 219
`d` option 68, 98, 132, 160, 194, 211, 215, 219, 223
date and time 7
default target 4
definition file, module 92, 119, 181
`delta` command (SCCS) 205
dependencies 5, 8, 17
dependencies, example of 14
dependency and commands 19
dependency, direct 5
dependency example, direct 5, 48
dependency example, indirect 5
dependency, indirect 5
dependency tree 6, 18, 19
`depends` makefile for 291, 295
`depends` utility 81, 144, 269
`depends.c` 273
direct dependency 5
direct dependency example 5, 48
directive, `elif` 100, 126, 162
directive, `if` 73, 100, 125, 126, 136, 162
directive, `if...else` 73, 100, 103, 125, 126, 137, 162, 165
directive, `ifdef` 72, 74, 125, 136, 138
directive, `include` 72, 99, 135, 161
directive, nested `if` 105, 167
directive, `undef` 100, 162
directives, dot 27, 42
directories, changing 46, 48, 195
DOS `errorlevel` statement 87, 114
DOS MAKE 67, 97
dot directives 27, 42

`e` option 57, 68, 98, 129, 132, 160, 190, 194, 211, 215, 220, 223
`e` option, `get` (SCCS) 205
`elif` directive 100, 126, 162
empty `AFLAGS` 33
empty `CFLAGS` 35
empty macro 34, 35

enhancing readability 69, 133
environment variable 52, 53, 54, 75, 138
environment variable, creating an 53
error messages 61
`errorlevel` statement 87, 114, 150, 176
errors, example of 61
example of makefile 2, 15, 22, 25, 32, 37, 39, 46, 48, 49, 52
executable files in UNIX 35
explicit commands 5
extension, file name 29
extensions and ~ 200

`f` option 12, 13, 19, 61, 68, 98, 132, 160, 194, 211, 215, 220, 223
file, `.exe` 7
file, autoresponse 85, 112, 174
file, module definition 92, 119, 181
file name extension 29
file name suffix 29
file names and macros 48
file, response 85, 112, 174
files, `.obj` 7
files in SCCS 200
files, UNIX 35
files, up-to-date 7
first makefile 2
first target 4
first target, making the 12

`get` command (SCCS) 203
`get`, `e` option (SCCS) 205
graphical user interface (GUI) 91, 118, 180, 206

`h` option 98, 160, 220
hard-coded commands 13, 14, 19
header files 80, 143
`help` option 68, 132, 211, 215

`I` option 98, 160, 220
`i` option 28, 42, 68, 87, 98, 114, 132, 150, 160, 176, 194, 212, 215, 220, 223
IBM Resource Compiler 154
IBM Resource Compiler (RC) 180
IBM's NMAKE 24, 34, 215
IBM's NMAKE on OS/2 131
IBM's NMAKE, precedence of macros in 152, 190
`icc` command 24, 34, 41, 147
IDE (interactive development environment) 97, 159
`if` directive 73, 100, 125, 126, 136, 162
`if` directive, nested 105, 167
`if...else` directive 73, 100, 103, 125, 126, 137, 162, 165
`if...else` example 74, 75, 137, 138
`if[...]` example 75, 138
`ifdef` directive 72, 74, 125, 136, 138
implicit command 30, 42
implicit target 30, 42
`include` directive 72, 99, 135, 161

include files 80, 143
indenting command lines 3, 8
indirect dependency 5
indirect dependency example 5
inference rule .asm.obj 33
inference rule .c.o 35, 200
inference rule .c.o: 224
inference rule .c.obj 30, 34, 35
inference rule .c: 201, 224
inference rule .c~.o: 203, 204, 224
inference rule .s.o 201
inference rule .s~.o: 204, 224
inference rule, predefined 34, 200
inference rule, user-defined 32, 43
inference rules 29, 30, 31, 32, 42
information substitution, macro 58
INIT environment variable 90, 152
interactive development environment (IDE) 97, 159
invoking Borland's MAKE (DOS) 97
invoking Borland's MAKE (OS/2) 159
invoking *make* 4, 6, 8, 11, 11, 211, 215, 219, 223
invoking *make* recursively 45
invoking *make* with a target 4, 9
invoking NMAKE 211, 215
invoking nmake 11
invoking NMAKE (DOS) 68
invoking NMAKE (OS/2) 132
invoking UNIX MAKE 193

K option 98, 109, 113, 128, 160, 171, 176, 220
k option 68, 212

link command 84
link386 command 147, 148, 149

m option 68, 99, 161, 212, 220
macro $@ 89, 116
macro, adding extension to 58
macro AFLAGS 33
macro AS 33
macro CC 15, 16, 41
macro CFLAGS 15, 16
macro, current target 24, 25
macro, empty 34, 35
macro, example of a 15
macro information substitution 58
macro names 22
macro names and ( ) 26
macro on the command line 49, 50
macro, using a 15
macro, value of a 15, 20, 74, 104, 137, 166
macro, what is a 15
macros 13, 20
macros, advantages of using 16, 20
macros and $() 15
macros and file names 48
macros, built-in 22
macros, commands with 14
macros in Borland's MAKE, precedence of 118, 129, 179, 190
macros in IBM's NMAKE, precedence of 152, 190

macros in Microsoft's NMAKE, precedence of 90, 128
macros, precedence of 207, 208
macros, predefined 22, 23, 31, 40
macros, printing 12
macros, user-defined 22, 23, 40
macros, value of *make* 26
MAKE (DOS), invoking Borland's 97
MAKE (OS/2), invoking Borland's 159
MAKE and ! 99, 125, 161
make and recursion 45, 47
MAKE, Borland's 24, 34, 219
make, invoking 4, 6, 8, 11, 11, 211, 215, 219, 223
MAKE, invoking UNIX System's 193
make, invoking with a target 4
MAKE on DOS 67, 97
MAKE on DOS (Borland's) 97
MAKE on OS/2 159
MAKE on OS/2 (Borland's) 159
MAKE on UNIX 193
MAKE options 98, 160, 194, 219, 223
make options 11, 12, 68, 98, 132, 160, 194, 211, 215, 219, 223
`MAKE` predefined macro 46, 82, 145, 197
make, reinvoking 48, 49
make, specifying file name 12
make, the theory 2
MAKE, UNIX system's 24, 193, 223
MAKE, UNIX's 35
MAKE: makefile for `depends` 295
MAKE: makefile for PM example 256
MAKE: makefile for `pmsamp` 256
MAKE: makefile for Windows example 239
MAKE: makefile for `winsamp` 239
MAKE: makefile for X Windows example 268
MAKE: makefile for `xmsamp.c` 268
MAKE: `pmsamp`, makefile for 256
MAKE: `winsamp`, makefile for 239
MAKE: `xmsamp.c`, makefile for 268
makefile 1, 6, 11, 12, 13
makefile example 2, 15, 22, 25, 32, 37, 39, 46, 48, 49, 52
makefile, first 2
makefile for `depends` 291, 295
makefile for `depends` (MAKE) 295
makefile for `depends` (NMAKE) 291
makefile for PM example 236, 253, 256
makefile for `pmsamp` 253, 256
makefile for Windows example 239
makefile for `winsamp` 236, 239
makefile for X Windows example 268
makefile for `xmsamp.c` 268
makefile name 12
makefile, renaming a 12
`MAKEFLAGS` predefined macro 197, 207
making the first target 12
`masm` 33
Microsoft Resource Compiler 92, 119, 180
Microsoft's C compiler 7
Microsoft's NMAKE 23, 34, 35, 67, 211
Microsoft's NMAKE, precedence

of macros in 90, 128
module definition 154
module definition file 92, 119, 181

N option 99, 161, 220
n option 12, 13, 19, 26, 40, 68, 99, 132, 161, 194, 197, 212, 216, 220, 223
name for a makefile 12
negation expression, ! 77, 140
nested if directives 105, 167
newline 79, 127
NMAKE (DOS), invoking 68
NMAKE (OS/2), invoking 132
NMAKE and ! 69, 125, 133
NMAKE, IBM's 24, 34, 215
NMAKE, invoking 211, 215
nmake, invoking 11
NMAKE, Microsoft's 23, 34, 35, 211
NMAKE on DOS (Microsoft's) 67
NMAKE on OS/2 131
NMAKE on OS/2 (IBM's) 131
NMAKE options 68, 132, 211, 215
NMAKE, preprocessing directives in 69, 133
NMAKE: makefile for depends 291
NMAKE: makefile for PM example 253
NMAKE: makefile for pmsamp 253
NMAKE: makefile for Windows example 236
NMAKE: makefile for winsamp 236
NMAKE: pmsamp, makefile for 253
NMAKE: winsamp, makefile for 236
nologo option 69, 132, 212, 216
NOT operator ! 104, 126, 166, 187
null suffix rule 201

object files in UNIX 35
option ? 68, 98, 132, 160, 211, 215, 219
option a 68, 98, 109, 110, 132, 160, 171, 211, 215, 219
option B 98, 160, 219
option b 68, 211
option c 68, 132, 201, 211, 215
option D 98, 102, 118, 160, 164, 179, 219
option d 68, 98, 132, 160, 194, 211, 215, 219, 223
option e 57, 68, 98, 129, 132, 160, 190, 194, 211, 215, 220, 223
option e (SCCS) 205
option f 12, 13, 19, 61, 68, 98, 132, 160, 194, 211, 215, 220, 223
option h 98, 160, 220
option help 68, 132, 211, 215
option I 98, 160, 220
option i 28, 42, 68, 87, 98, 114, 132, 150, 160, 176, 194, 212, 215, 220, 223
option indicator (-) 12, 68, 98, 132, 160, 194
option indicator (/) 12, 68, 98, 132, 160
option K 98, 109, 113, 128, 160, 171, 176, 220
option k 68, 212
option m 68, 99, 161, 212, 220

option N 99, 161, 220
option n 12, 13, 19, 26, 40, 68, 99, 132, 161, 194, 197, 212, 216, 220, 223
option nologo 69, 132, 212, 216
option p 12, 26, 36, 40, 69, 99, 132, 161, 194, 197, 201, 212, 216, 220, 223
option q 69, 132, 194, 212, 216, 223
option r 69, 90, 99, 118, 128, 133, 152, 161, 189, 194, 212, 216, 220, 224
option S 99, 161, 220
option s 28, 42, 69, 99, 133, 161, 194, 212, 216, 220, 224
option t 69, 133, 194, 212, 216, 224
option U 99, 161, 220
option v 69, 212
option W 99, 161, 220
option x 69, 133, 212, 216
option z 69, 212
options, Borland's MAKE 98, 160, 219
options, MAKE 98, 160, 194, 219, 223
options, *make* 11, 12
options, NMAKE 68, 132, 211, 215
options, order of 13, 19
options, UNIX MAKE 194, 223
order of options 13, 19
OS/2 errorlevel statement 150, 176
OS/2 MAKE 131, 159
OS/2 NMAKE 131
p option 12, 26, 36, 40, 69, 99, 132, 161, 194, 197, 201, 212, 216, 220, 223
parentheses and macro names 26
PC command line indentation 3
placement, target/dependency line 3
pmsamp, makefile for 253, 256
pmsamp.c 243
pmsamp.dlg 252
pmsamp.rc 251
position of a target 19
pound sign (#) 3
precedence of macros 90, 118, 128, 129, 152, 179, 190, 207, 208
precedence of macros in Borland's MAKE 118, 129, 179, 190
precedence of macros in IBM's NMAKE 152, 190
precedence of macros in Microsoft's NMAKE 90, 128
predefined inference rule 34, 200
predefined macro $& 41, 50
predefined macro $* 24, 25, 32, 41
predefined macro $< 30, 41, 201
predefined macro $@ 24, 25, 33, 41
predefined macro $d 101, 102, 126, 163, 164, 187
predefined macro CC 23, 24, 25
predefined macro MAKE 46, 82, 145, 197
predefined macro MAKEFLAGS 197, 207
predefined macros 22, 23, 31, 40
predefined pseudotargets 27, 42

preprocessing directives in MAKE 99, 161
preprocessing directives in NMAKE 69, 133
preprocessor directive, ! 69, 77, 125, 133, 140
printing macros 12
prj2mak 97
program depends 81, 144, 269
pseudotarget .IGNORE: 28, 42, 87, 150
pseudotarget, .ignore: 114, 176
pseudotarget .SILENT: 28, 42
pseudotarget .SUFFIXES: 28, 42, 62, 200
pseudotarget .suffixes: 28, 50, 51, 56, 62, 108, 123, 171, 185
pseudotargets 27, 42, 47, 83, 146
pseudotargets, predefined 27, 42
pseudotargets, user-defined 27, 42

q option 69, 132, 194, 212, 216, 223

r option 69, 90, 99, 118, 128, 133, 152, 161, 189, 194, 212, 216, 220, 224
RC, IBM Resource Compiler 154, 180
RC, Microsoft Resource Compiler 92, 119, 180
readability 3, 99, 161
readability, enhancing 69, 133
recursion and *make* 45, 47
reinvoking *make* 48, 49
renaming a makefile 12
Resource Compiler (RC) 92, 119, 154, 180
response file 85, 112, 174
return code 28
rules, inference 29, 30, 31, 32, 42
rules, suffix 29, 42

S option 99, 161, 220
s option 28, 42, 69, 99, 133, 161, 194, 212, 216, 220, 224
SCCS files 200
SCCS: admin command 202
SCCS: delta command 205
SCCS: get command 203
set command 52, 75, 138
spaces and commands 194
specifying a target 18
specifying file name in *make* 12
statement, errorlevel 87, 114, 150, 176
substitution, macro information 58
suffix, file name 29
suffix, null 201
suffix rules 29, 30, 42
synonym for current target 24
synonym for target without extension 24
syntax rules 2

t option 69, 133, 194, 212, 216, 224
tabs and commands 194
target, default 4
target, example of 4, 14, 24

target, implicit 30, 42
target line and : 4
target macro, current 24
target, position of a 19
target, specifying a 18
target, the first 4
target, what is 4
target without an extension 24
target without extension, synonym for 24
target/dependency line placement 3
targets and commands 5, 13
theory of *make* 2
time and date 7
time stamp 2, 7, 8
`tlink` command 111, 119, 173, 180
`tools.ini` file 67, 128, 131
tree, dependency 6, 18, 19

`U` option 99, 161, 220
`undef` directive 100, 162
UNIX, `.o` files in 35, 200
UNIX, C files in 35
UNIX, executable files in 35
UNIX files 35
UNIX MAKE, example of 51
UNIX MAKE options 194, 223
UNIX, object files in 35
UNIX system's C compiler 16
UNIX system's *make* 24, 193, 223
UNIX systems command line indentation 3
UNIX's MAKE 35
up-to-date files 7
user-defined inference rules 32, 43
user-defined macros 22, 23, 40
user-defined pseudotargets 27, 42
using a macro 15
using `set` 52, 75, 138
utility `depends` 81, 144, 269

`v` option 69, 212
value of a macro 15, 20, 74, 104, 137, 166
value of *make* macros 26
variable, environment 52, 53, 54

`W` option 99, 161, 220
`winsamp`, makefile for 236, 239
`winsamp.c` 227
`winsamp.dlg` 235
`winsamp.rc` 234

`x` option 69, 133, 212, 216
X Windows 206, 207
`xmsamp.c` 259
`xmsamp.c`, makefile for 268
`xwindows.mak` 206

`z` option 69, 212

UNIX
Utilities and Tools

SECOND EDITION

MASTERING MAKE

A GUIDE TO BUILDING PROGRAMS ON DOS, OS/2®, AND UNIX® SYSTEMS

CLOVIS TONDO / ANDREW NATHANSON / EDEN YOUNT

A practical introduction to ***make***, this comprehensive tutorial shows programmers how to exploit the full power of this valuable utility. Whether you are programming in a DOS, or OS/2, or UNIX system environment, this guide explains in logical, easy-to-follow steps how ***make*** works and shows how to use it in a variety of applications.

Key features:

- Combines elements of both a tutorial and a reference book, assuming little or no experience with the ***make*** command
- Explains how to create GUI applications using ***make*** on X Windows, Presentation Manager, and Windows
- Offers numerous examples of tested, flexible, and easy-to-use **makefiles** that you can apply to your own software development projects
- Helps you avoid the unproductive process of learning ***make*** by trial and error
- Examines the specifics of popular versions of ***make***, including **Microsoft's NMAKE**, **Borland's MAKE**, and **MAKE for UNIX Systems**

About the authors:

Clovis Tondo is the co-author of several programming books, including *The C Answer Book*, *Writing DOS Device Drivers in C*, *Writing UNIX® Device Drivers in C*, and *Data Structures and Program Design in C*.

Andrew Nathanson is an independent software engineer involved in application and operating system development for over 18 years.

Eden Yount is a technical writer with over 10 years of experience in the computer industry and a member of the Society for Technical Communication.

ISBN 0-13-121906-5

9 780131 219069 90000

Prentice Hall
Englewood Cliffs, NJ 07632